Agustín Riscos-Núñez  Claudio Zandron
Gexiang Zhang  David Orellana Martín
(Eds.)

# Membrane Computing

International Conference on Membrane Computing 2026

Seville, Spain, September 1–4, 2026

Proceedings



**Membrane Computing**
International Conference on Membrane Computing 2026
Seville, Spain, September 1–4, 2026
Proceedings

**ISBN:** 978-84-09-91134-9

Copyright in the individual contributions © 2026 the respective authors.
All rights remain with the authors.

# Preface

This volume contains the contributions presented at ICMC26, the International Conference on Membrane Computing 2026, held in Seville, Spain, during September 1–4, 2026. Additional information can be found at the website of the conference, at the address: `www.gcn.us.es/icmc2026`

The CMC series started by Gheorghe Păun in Curtea de Argeș, Romania, in 2000, with three editions of the Workshop on Membrane Computing, WMC. After that, the workshops were held in Tarragona, Spain (2003), Milan, Italy (2004), Vienna, Austria (2005), Leiden, The Netherlands (2006), Thessaloniki, Greece (2007), and in Edinburgh, UK (2008).

The tenth edition was organized again in Curtea de Argeș, in 2009, when it was decided to continue the series as the Conference on Membrane Computing (CMC). The following editions were held in Jena, Germany (2010), Fontainebleau, France (2011), Budapest, Hungary (2012), Chișinău, Moldova (2013), Prague, Czech Republic (2014), Valencia, Spain (2015), Milan, Italy (2016), Bradford, UK (2017), Dresden, Germany (2018). The 20th Conference on Membrane Computing returned to Curtea de Argeș in August 2019. The 21st edition was celebrated online organized in Vienna, Austria (2020). The next editions were held in Debrecen, Hungary (2021), Trieste, Italy (2022), Opava, Czech Republic (2023) and Nice, France (2024).

A regional version of CMC, the Asian Conference on Membrane Computing, ACMC, started with the 2012 edition in Wuhan, China. The following editions were held in Chengdu, China (2013), Coimbatore, India (2014), Anhui, China (2015), Selangor, Malaysia (2016), Chengdu, China (2017), Auckland, New Zealand (2018), and Xiamen, China (2019). The 9th edition was celebrated online organized in Ulaanbaatar, Mongolia, joint with the CMC. The next editions were held in Chengdu, China (2021), Quezon City, The Philippines (2022), Chengdu, China (2023) and Singapore (2024).

From 2025, the P community joined forces and proposed the celebration of the International Conference on Membrane Computing (ICMC), to be held alternately in Europe and Asia/Oceania. The 1st edition was celebrated in 2025 in Chengdu, China.

ICMC26 was held at the University of Seville, organized by the members of the Research Group on Natural Computing, mostly known in the P community as the "Sevilla Team".

In this edition, four researchers were invited to deliver invited talks, providing different perspectives of theory and applications in the area of Natural Computing: Raluca Lefticaru from the University of Bradford (Bradford, UK), Ángel Goñi-Moreno from the Biocomputation Laboratory, CNB-CSIC (Madrid, Spain), Prithwineel Paul from the Institute of Engineering & Management (Kolkata, India), and Tao Wang from the Xihua University (Chengdu, China).

ICMC26 was a *special* edition, as it has been co-located with the Honorary Scientific Day in Memory of Prof. Mario J. Pérez-Jiménez. Mario J. Pérez-Jiménez (13 November 1948 - 17 September 2025) was the founder of the Research Group on Natural Computing, and one of the most recognized researchers in the area of Membrane Computing. Moreover, he was also distinguished as a top researcher in different areas such as artificial intelligence, image vision and others.

Mario was a Full Professor and an Emeritus Professor of the Department of Computer Science and Artificial Intelligence of the University of Seville. He was Honorary Professor of Huazhong University of Science and Technology in Wuhan, China. Besides, he was the first Andalusian researcher to be invited to join the Informatics section of the Academia Europaea. He received the FAMA award of the University of Seville in the area of Engineering and Architecture in 2014. He was the first scientist to receive the Important Contributions to Membrane Computing award under the auspices of the European Molecular Computing Consortium. Apart from that, he received some awards for his excellent scientific trajectory in the area of Natural Computing and, more precisely, Membrane Computing. The group has organized 20 editions of the Brainstorming Week on Membrane Computing, and along the years has hosted both young PhD students as well as senior researchers from all around the World, making Seville being usually called "the Mecca of Membrane Computing".

He is one of the pioneers of Membrane Computing, creating a computational complexity theory in this framework in which more than 500 research papers are based. He is the creator of a new methodology that develops unconventional techniques to tackle the P vs NP problem. He published more than 400 scientific documents in collaboration with more than 150 national and international co-authors. He supervised 15 PhD theses, 4 of them distinguished with the Extraordinary PhD award from University of Seville.

We would like to thank all the participants, both on-site and online, for their valuable contributions, active participation, and for helping to make the conference a success. We also extend our thanks to the invited speakers, session chairs, reviewers, and members of the Steering Committee, the Organizing Committee and the Program Committee for their dedication and support throughout the event. We would like to acknowledge the institutional support received from the School "Escuela Técnica Superior de Ingeniería Informática (ETSII)", the Research Institute of Computer Engineering (I3US), and the Smart Computer Systems Research and Engineering Lab (SCORE).

September 2026

David Orellana-Martín
Agustín Riscos-Núñez
Claudio Zandron
Gexiang Zhang

# Organization

## Steering Committee

| | |
|---|---|
| Henry Adorna | Quezon City, Philippines |
| Artiom Alhazov | Chișinău, Moldova |
| Bogdan Aman | Iași, Romania |
| Matteo Cavaliere | Manchester, UK |
| Erzsébet Csuhaj-Varjú | Budapest, Hungary |
| Giuditta Franco | Verona, Italy |
| Rudolf Freund | Wien, Austria |
| Marian Gheorghe | Bradford, UK – Honorary member |
| Thomas Hinze | Jena, Germany |
| Florentin Ipate | Bucharest, Romania |
| Shankara N. Krishna | Bombay, India |
| Alberto Leporati | Milan, Italy |
| Ferrante Neri | Nottingham, UK |
| Taishin Y. Nishida | Toyama, Japan |
| Linqiang Pan | Wuhan, China – Co-chair |
| Gheorghe Păun | Bucharest, Romania – Honorary member |
| Agustín Riscos-Núñez | Sevilla, Spain |
| Jose M. Sempere | Valencia, Spain |
| Petr Sosík | Opava, Czech Republic |
| Kumbakonam Govindarajan Subramanian | Chennai, India |
| György Vaszil | Debrecen, Hungary |
| Sergey Verlan | Paris, France |
| Claudio Zandron | Milan, Italy – Co-chair |
| Gexiang Zhang | Chengdu, China |

## Organizing Committee

| | |
|---|---|
| Agustín Riscos Núñez | Seville, Spain – Chair |
| Gheorghe Păun | Seville, Spain – Honorary chair |
| José Antonio Andreu Guzmán | Seville, Spain |
| Carmen Graciani | Seville, Spain |
| Miguel Ángel Martínez del Amor | Seville, Spain |
| Gonzalo Martínez Fernández | Seville, Spain |
| David Orellana Martín | Seville, Spain |
| Antonio Ramírez de Arellano Marrero | Seville, Spain |
| Álvaro Romero Jiménez | Seville, Spain |

| | |
|---|---|
| Luis Valencia Cabrera | Seville, Spain |

## Program Committee

| | |
|---|---|
| Henry Adorna | Quezon City, Philippines |
| Artiom Alhazov | Chișinău, Moldova |
| Bogdan Aman | Iași, Romania |
| Péter Battyányi | Debrecen, Hungary |
| Francis G.C. Cabarle | Quezon City, Philippines |
| Lucie Ciencialová | Opava, Czech Republic |
| Erzsébet Csuhaj-Varjú | Budapest, Hungary |
| Rudolf Freund | TU Wien, Austria |
| Giuditta Franco | Verona, Italy |
| Zsolt Gazdag | Szeged, Hungary |
| Marian Gheorghe | Bradford, UK |
| Ping Guo | Chongqing, China |
| Juanjuan He | Wuhan, China |
| Thomas Hinze | Jena, Germany |
| Florentin Ipate | Bucharest, Romania |
| Shankara N. Krishna | Bombay, India |
| Tseren-Onolt Ishdorj | Ulaanbaatar, Mongolia |
| Savas Konur | Bradford, UK |
| Alberto Leporati | Milan, Italy |
| Jia Li | Chongqing, China |
| Xiyu Liu | Jinan, China |
| Ravie Chandren Muniyandi | Bangi, Malaysia |
| Ferrante Neri | Nottingham, UK |
| Radu Nicolescu | Auckland, New Zealand |
| Taishin Nishida | Toyama, Japan |
| Yunyun Niu | Beijing, China |
| David Orellana-Martín | Sevilla, Spain |
| Linqiang Pan | Wuhan, China |
| Andrei Păun | Bucharest, Romania |
| Gheorghe Păun | Bucharest, Romania |
| Hong Peng | Chengdu, China |
| Antonio E. Porreca | Marseille, France |
| Agustín Riscos-Núñez | Sevilla, Spain – Co-chair |
| Haina Rong | Chengdu, China |
| Jose M. Sempere | Valencia, Spain |
| Bosheng Song | Changsha, China |
| Tao Song | Qingdao, China |
| Petr Sosík | Opava, Czech Republic |
| K.G. Subramanian | Chennai, India |
| D.G. Thomas | Chennai, India |
| György Vaszil | Debrecen, Hungary |
| Sergey Verlan | Paris, France |

| | |
|---|---|
| Jun Wang | Chengdu, China |
| Tingfang Wu | Suzhou, China |
| Jianhua Xiao | Tianjin, China |
| Jie Xue | Jinan, China |
| Hsu-Chun Yen | Taiwan, R.O.C |
| Jianying Yuan | Chengdu, China |
| Claudio Zandron | Milan, Italy – Co-chair |
| Xiangxiang Zeng | Xiamen, China |
| Gexiang Zhang | Chengdu, China – Co-chair |
| Ming Zhu | Chengdu, China |

# Table of Contents


**Invited Talks** **1**

Living cells as operating systems for programmable genetic circuits . . . . . 3
*Ángel Goñi-Moreno*

Kernel P Systems: Overview, Tools, and Recent Developments . . . . . 5
*Raluca Lefticaru*

Quantum-Membrane Neural Systems . . . . . 7
*Prithwineel Paul*

Recent Applications of Membrane Computing in Intelligent Power Systems: Case Studies from Our Research Group . . . . . 9
*Tao Wang*

**Regular Talks** **11**

A Weighted Spiking Neural P Systems Approach to Distributed SYN Flood Detection . . . . . 13
*Jacky John D. Abucay, Shane I. Odhuno, Kelvin C. Buño, Francis George C. Cabarle*

Numerical Networks of Cells with Forward Activation and Blocking of Rules . . . . . 43
*Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, Sergey Verlan*

A matrix-based and GPU-accelerated simulator for ASNP systems . . . . . 67
*Matteo Balzerani, Sandro Erba, Giuditta Franco, Claudio Zandron*

Reaction systems with rough sets . . . . . 83
*Péter Battyányi*

Probabilistic Verification of Robot Controllers using Enzymatic Numerical P Systems Models . . . . . 99
*Radu Traian Bobe, Florentin Ipate, Savas Konur*

Reducing capacity in P Colonies . . . . . 127
*Lucie Ciencialová, Luděk Cienciala*

Scalable Verification-Aware Kernel P Systems: Indexed Modelling, Candidate Generation, and Trace-Level Verification . . . . . 145
*Doru Constantin, Cristina Tudose, Ștefan Iacob, Ionuț Mihai Niculescu, Florentin Ipate, Marian Gheorghe*

A polynomial-time solution to the latin square completion problem by means of P systems with active membranes ..... 163
*Jerónimo Crespo, David Orellana-Martín*

A Visual Discrete Event-based Simulator for the Optimization of Plant Defense Strategies against Herbivores as Application of a Tissue P System 179
*Lucas Dietrich, Benjamin Förster, Peter Langendörfer, Thomas Hinze*

Control Languages Associated with Cell-like Spiking Neural P Systems .. 207
*Ajeesh Ramanujan, Atulya Nagar, Rodica Ceterchi, K.G. Subramanian, Marian Gheorghe*

Spiking Neural P Systems in kPWorkbench ..... 225
*Ștefan Iacob, Florentin Ipate, Marian Gheorghe, Ionuț Mihai Niculescu*

A Membrane Computing-Based Model for Image Classification ..... 255
*Katalin Anna Lázár, Leóna Anasztázia Rajhona*

Solving 3-SAT via Enzymatic Numerical P Systems with Membrane Division ..... 283
*Alberto Leporati*

Toward Partition-Aware Design of Spiking Neural P Systems: A Bulk Synchronous Parallel Cost Model and a Binary Multiplication Case Study ..... 309
*Job J. Lipat, Henry N. Adorna, Francis George C. Cabarle, Miguel Ángel Martínez-del-Amor, David Orellana-Martín*

Small universal multiset reaction systems ..... 331
*Annemarie-Beatrix Messner, Andrei Păun*

Algebraic Topological Data Analysis of Membrane Systems via Multi-Hypergraph Incidence Models ..... 345
*Pedro Real*

Towards olive tree water stress classification with spiking neural P systems 361
*David Orellana-Martín, Agustín Riscos-Núñez, Alejandro Linares-Barranco*

Program-oriented models of computation and virus machines ..... 375
*David Orellana-Martín, Luis Valencia-Cabrera, Mario J. Pérez-Jiménez*

The power of Virus Machines: generating strings ..... 393
*Antonio Ramírez-de-Arellano-Marrero, David Orellana-Martín, Francis George C. Cabarle*

An assembly P system which generates finite approximations of the Hilbert curve ..... 407
*Taishin Y. Nishida*

Spiking Neural P systems with colored spikes and multiple channels ..... 425
*José M. Sempere, Rodrigo Llanes*

Numerical 2D P Colonies as a Data Management Architecture for Biological Computers ..... 447
*Daniel Valenta, Miroslav Langer*

TENNCell: Transformation and Execution of Numerical Networks of Cells 465
*Sergey Verlan, Junying Xiao, Gexiang Zhang, Qiyu Liu*

**Author Index** ..... 481

# Invited Talks

# Living cells as operating systems for programmable genetic circuits

Ángel Goñi-Moreno[0000−0002−2097−2507]

Systems Biology Department, Centro Nacional de Biotecnología (CNB) CSIC, Darwin 3, Madrid 28049, Spain
angel.goni@cnb.csic.es

**Abstract.** The design and implementation of genetic circuits is transforming cellular computing into a highly active engineering discipline. By arranging genetic components into pre-defined regulatory cascades, we can process inputs into predictable outputs—enabling living cells to execute Boolean logic, where specific input combinations switch target proteins ON or OFF. However, these circuits do not run in isolation; the host cell acts as a biological operating system, defining contextual dependencies that constrain circuit behavior. Host-specific mechanisms, such as resource allocation, impose strict performance limits on the genetic software running within them. Crucially, we can turn this limitation into an engineering advantage: instead of altering a circuit's DNA sequence, we can reprogram its functional behavior simply by modifying its cellular context. In this talk, I highlight the host context not as a barrier, but as a dynamic, programmable computational resource essential for scaling up complex biocomputations.

# Kernel P Systems: Overview, Tools, and Recent Developments

Raluca Lefticaru

School of Computer Science and Electronics, University of Bradford
West Yorkshire, BD7 1DP, United Kingdom
r.lefticaru@bradford.ac.uk

**Abstract.** Introduced in 2012, kernel P systems aim to unify the most used concepts from P systems into a single, coherent model. This talk will provide an overview of kernel P systems, with emphasis on their expressive power, modelling capacity, and relationships to other membrane-computing formalisms, including spiking P systems. It will then discuss the specification languages and software tools that support modelling, simulation and verification of these models, with a focus on kP-Lingua and kPWorkbench. Some case studies, applications of kernel P systems and recent developments will be presented, followed by open problems, and opportunities for future research and collaborations.

# Quantum-Membrane Neural Systems

Prithwineel Paul[0000−0001−8351−3407]

Department of Computer Science and Engineering
Institute of Engineering and Management, Kolkata, India
prithwineel.paul@iem.edu.in

**Abstract.** Quantum-Membrane Neural Systems (QMNS) represent a novel computational paradigm that integrates principles of membrane computing with the computational advantages of quantum computing and neural networks. In this framework, the properties of spiking membrane systems and superposition and entanglement properties of quantum circuits are exploited to enhance feature representation and learning efficiency. By combining bio-inspired computation, quantum information processing, and deep learning, QMNS aims to develop scalable, robust, and energy-efficient intelligent systems for complex data analysis. The proposed paradigm has the potential to address limitations of conventional deep learning in terms of parallelism, interpretability, and computational efficiency. Future applications include medical image analysis, quantum communication networks, and next-generation artificial intelligence. Quantum–Membrane Neural Systems open a promising interdisciplinary research direction at the intersection of membrane computing, quantum machine learning, and bio-inspired artificial intelligence.

# Recent Applications of Membrane Computing in Intelligent Power Systems: Case Studies from Our Research Group

Tao Wang[0000−0002−2097−2507]

School of Electrical Engineering and Electronic Information
Xihua University, Chengdu 610039, China
twang@mail.xhu.edu.cn

**Abstract.** Membrane computing, a bio-inspired computational paradigm, has shown great potential in solving complex optimization, prediction, and fault diagnosis problems. Our research group has been actively exploring its applications in intelligent power systems. This presentation will introduce our recent research achievements over the past year, including transmission line icing prediction, short-term power load forecasting, photovoltaic fault diagnosis, and distribution network fault section localization. Focusing on practical engineering applications, the talk will present representative case studies and experimental results to demonstrate the effectiveness of membrane computing in intelligent power systems and discuss its future application prospects.

# Regular Talks

# A Weighted Spiking Neural P Systems Approach to Distributed SYN Flood Detection

Jacky John D. Abucay, Shane I. Odhuno, Kelvin C. Buño, and Francis George C. Cabarle

Department of Computer Science, University of the Philippines Diliman, Quezon City, Metro Manila, Philippines
{jdabucay,siodhuno,fccabarle,kcbuno}@up.edu.ph

**Abstract.** SYN flood attacks remain a prevalent form of distributed denial-of-service in modern networks, exploiting the TCP three-way handshake to exhaust server resources. Traditional detection methods often struggle to balance adaptability, speed, and accuracy, particularly against modern botnet-driven attacks that originate from many compromised hosts. Therefore, this paper proposes a SYN Flood Detection System (SFDS) based on weighted Spiking Neural P (SN P) systems, as their temporal and parallel characteristics suit network-traffic analysis. Specifically, the proposed SFDS monitors three features: the arrival rate of SYN packets, the per-source ratio of SYN packets to ACK packets, and the count of anomalous (i.e., spoofed) SYNs, raising an alarm when at least two features indicate an attack. Its per-source design lets the system process multiple concurrent input streams in parallel and isolate a flooding source that aggregate statistics would dilute away. At its core, an end-of-stream protocol and a division-free integer encoding allow the thresholded SYN:ACK proportion to be evaluated exactly over variable-length streams, all while every neuron's spike count remains non-negative. Finally, rather than fixing the feature thresholds by hand, we calibrate them with a sweep over eight labeled scenarios of mixed synthetic traffic: the detector reproduces every scenario's ground-truth label, and each feature separates its target behavior across a wide threshold band.

**Keywords:** Spiking Neural P Systems · SYN Flood Detection · Parallel Computing · Membrane Computing · Cybersecurity

## 1 Introduction

Distributed Denial-of-Service (DDoS) attacks are among the most significant cybersecurity threats of the digital era [20]. These attacks overwhelm target systems by flooding them with malicious traffic, rendering services unavailable to legitimate users.

Among the various DDoS attack vectors, SYN flooding attacks in particular exploit the TCP three-way handshake mechanism by initiating numerous half-open connections that exhaust server resources. The consequences of such attacks

can be severe; for instance, in healthcare, they can slow or even shut down entire systems, putting the lives of patients at risk [5].

However, mitigating DDoS attacks is challenging because they can take on many different forms [20], and traditional detection methods often struggle to balance adaptability, speed, and accuracy.

Moreover, modern DDoS attacks are increasingly orchestrated by large botnets, with traffic originating from many compromised hosts directed toward a single target [28]. A detector that observes only a single source, therefore, may only see a fraction of the attack, missing patterns that emerge only when signals from multiple sources are aggregated.

Given these realities, there is thus a need for novel approaches that can effectively detect SYN flood attacks, while also maintaining computational efficiency, flexibility, and scalability.

To address these challenges, this paper proposes the employment of Spiking Neural P (SN P) systems as a foundation for intrusion detection. SN P systems are membrane computing models inspired by how the human brain operates, leveraging neurons, synapses, spikes, and delays to process information. Their temporal and parallel characteristics make them particularly well-suited for analyzing network traffic patterns.

Therefore, this study makes the following contributions:

1. **Design:** We define a weighted SN P SYN Flood Detection System (SFDS) that operates on variable-length, multi-source input streams. At its core is a division-free integer encoding mechanism that evaluates a thresholded per-source SYN:ACK proportion exactly at the end of each stream. Every rule and initial spike content is derived by formula from the system's parameters. Moreover, all of this is achieved within the standard weighted SN P model, without fuzzy or learning extensions.
2. **Evaluation:** We also build a reproducible evaluation pipeline [1]. It consists of a multi-device traffic lab that produces genuine TCP captures, a windowing and spike-encoding stage, and a headless SN P system simulator. We then generate eight labeled traffic scenarios that exercise the three monitored features individually and in combination.
3. **Results:** Finally, we demonstrate that the SFDS reproduces the ground-truth label of all eight scenarios, including the combined majority-vote alarm. A threshold sweep further shows that each feature separates its target behavior across a wide band of thresholds. In other words, the feature thresholds can be calibrated systematically from labeled traffic rather than chosen by hand.

The proposed SN P SFDS monitors three features: `syn_packet_rate`, `syn_ack_ratio`, and `anomalous_syn_pattern`. These features are responsible for checking whether the SYN packets in a time window arrive faster than a threshold, whether any single source's proportion of SYNs to ACKs exceeds a threshold, and whether the count of anomalous (i.e., spoofed) SYNs is past a threshold, respectively. The system then flags an attack when two or more of

these features are triggered within the same window. More details about these features can be found in Definitions 8, 9, and 10.

Note that the scope of this work covers the design, verification, and empirical evaluation of the SFDS for SYN flood attacks specifically; other DDoS attack vectors (such as UDP and HTTP floods) as well as other detection-system variants (such as evolutionary-based approaches) are noted as directions for future work instead.

Positive results from this paper would carry significance on two fronts. For SYN flood detection, this study would demonstrate that formal computational models can serve as a viable alternative or complement to heuristic and machine-learning-based methods. This is because unlike learned detectors, every verdict of the SFDS is traceable to rule constants that are *derived* rather than trained (with provable guarantees such as **non-negativity** and **no premature firing**) and its thresholds can be changed without any retraining.

For SN P research, this study would extend the range of cybersecurity applications previously explored [22] to include the multi-source, variable-length case, broadening the practical relevance of membrane computing in network security.

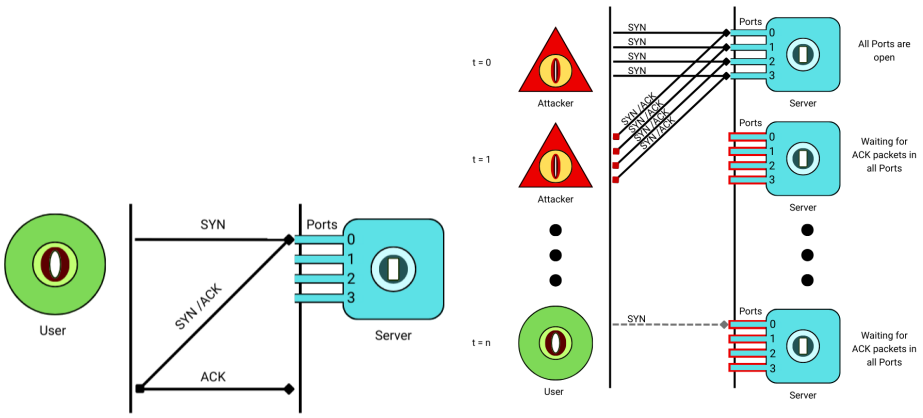
The overview of the entire paper is as follows: Section 2 reviews the related literature: Section 2.1 gives an overview of SYN flooding attacks, Section 2.2 covers existing flood detection systems, and Sections 2.3, 2.4, and 2.5 introduce SN P systems, their weighted variant, and their applications in cybersecurity. Section 3 then formalizes the SFDS. Specifically, Section 3.1 defines the detection problem and the three features, Section 3.2 defines the weighted SN P SFDS itself, Sections 3.3–3.5 describe how the traffic is generated, windowed, and encoded into spike trains, and Sections 3.6–3.11 walk through each layer of the system alongside the derivations of the detector constants. Section 3.12 then discusses how the threshold parameters may be modified, while Section 3.13 describes the simulation tool. Afterwards, Section 4 presents the results: Section 4.1 describes the eight traffic scenarios, while Sections 4.3 and 4.4 verify the detector at its default thresholds and calibrate each threshold empirically, and Section 4.5 situates the system among existing detectors. Lastly, Section 5 concludes the study by summarizing its contributions and outlining directions for future work.

## 2 Review of Related Literature

This section reviews the work that the SFDS builds on. In here, we cover flooding attacks (specifically TCP SYN flooding), existing SYN flood detection systems, SN P systems, weighted SN P systems, and the usage of SN P systems in the context of cybersecurity.

### 2.1 TCP SYN Flooding Attacks

A typical TCP handshake between a user and a server, shown in Figure 1a, consists of exchanging three packets: the SYN packet, the SYN-ACK packet,




(a) Normal TCP Handshake (b) SYN Flooding Attack

Fig. 1: TCP handshake behavior under (a) normal operation and (b) a SYN flooding attack.

and the ACK packet. The user first sends a SYN packet and the server then responds with the corresponding SYN-ACK packet. The server then waits for the user to send the remaining ACK packet [9]. However, if a user starts sending multiple TCP requests to the server, making the server busy, it could be a sign of a DoS attack.

One kind of malicious behavior is a SYN flooding attack. As shown in Figure 1b, it is a TCP based DoS method where the attacker transmits a large number of TCP SYN packets to their target server. The intended consequence, therefore, is to not complete the TCP three-way handshakes by making all of the ports unavailable for listening to new TCP SYN packets. This exhausts resources on the victim side and prevents the opening of legitimate connections [8].

### 2.2 SYN Flood Detection Systems

Scarfone and Mell [25] defined *intrusion detection* as the process of monitoring events within a computer system or network to determine whether certain behaviors or anomalies are considered a threat to the machine.

Haris et al. [13] proposed a SYN flood detection system using anomaly pattern checking that can be found on the network traffic. This was done by checking the IP and TCP headers, which contain a significant amount of information about the packets being received. By analyzing these fields, we will be able to know the behavior of the attack and respond accordingly.

Luckner [19] approached the problem of SYN flood detection with the use of Decision Trees and Deterministic Finite Automata (DFA). To start, the study first trained a Decision Tree on labeled NetFlow features. It then extracted each

root-to-attack leaf path as an ordered sequence of decision rules. Afterwards, each rule was encoded as an alphabet symbol and a Nondeterministic Finite Automaton (NFA) was constructed to accept those sequences. This NFA was then finally converted into a DFA and simulated online over a packet stream, outputting a SYN flood signal when it reaches an accepting state.

AlDossary et al. [2] opted to use a Time-Dependent Finite Automaton (TDFA) instead. In their paper, they built a four-component pipeline for SYN-flood detection: a data source, a packet filter, a token generator, and a TDFA. First, the data source was responsible for supplying TCP traffic from live captures and log files. Second, the packet filter parsed their captured data and extracted metadata from it. Third, a token generator was then used to convert each filtered event into ASCII tokens that encoded source IP, SYN count, and SYN inter-arrival time. Finally, a TDFA consumes the token stream and detects attacks. In their design, the automaton has three states (Q0, Q1, Q2) and raises a DoS alert if the measured inter-arrival time while in state Q1 exceeds ten seconds.

In addition to the aforementioned single-vantage-point approaches, there have also been prior works on *collaborative* or *multi-source* intrusion detection. The core insight is that distributed attacks, such as those launched by botnets, leave detection-relevant patterns that no single sensor can fully observe [27]. This, therefore, makes aggregating signals from multiple sources or input streams necessary for reliable detection.

For instance, Han et al. [12] proposed a Chord-protocol-based collaborative detection scheme in which edge-network monitors locally detect suspicious traffic. These monitors then aggregate their local detection results through a peer-to-peer "AutoTree" structure, with the root node outputting the final verdict. Their experimental results demonstrated that collaborative/distributed detection (CDD) obtained a 98% detection rate against the 73% detection rate of single-point detection (SPD).

As seen in the studies reviewed, the methodology for building intrusion detection systems, both at single observation points and across multiple coordinated ones, follows the pattern of gathering the data, preprocessing the data, building the machine, and analyzing the output. These studies have thus provided valuable insights into how we can build the SFDS. However, we require more information regarding the machine that we will mainly be focusing on, that is, the SN P system, which will be discussed in the following sections.

### 2.3 Spiking Neural P Systems

As mentioned, one model in particular will serve as the subject of interest for this study: a Spiking Neural P (SN P) system. First introduced by Ionescu et al. [16], SN P systems are membrane computing models inspired by how biological neurons communicate through electrical impulses called *spikes*.

Unlike traditional neural network models, SN P systems encode information not in sequences of different symbols, but in the *timing* of when a single type of signal (the spike) occurs [18]. Notably, certain variants of membrane computing

models (such as those with membrane division) have been shown to solve NP-complete problems in polynomial time with exponential space trade-offs [23]. The temporal and parallel characteristics of SN P systems are of particular interest to this anomaly detection study.

**Definition 1 (Syntax).** *Formally, an SN P system of degree $m \geq 1$, where $m$ is the number of neurons, is defined in [16,17,6] as a tuple*

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out),$$

*where:*

- $O = \{a\}$ *is the singleton alphabet, where $a$ denotes a spike.*
- $\sigma_i = (n_i, R_i)$ *is the $i$-th neuron, for $1 \leq i \leq m$:*
  - $n_i \geq 0$ *is the initial number of spikes contained in the neuron.*
  - $R_i$ *is a finite set of rules of one of the following forms:*
    1. ***Spiking rule:*** $E/a^c \rightarrow a^p; d$*, where $E$ is a regular expression over $\{a\}$, $c, p, d \in \mathbb{N}$ with $c \geq 1$ and $c \geq p$, and where $d$ is the delay.*
    2. ***Forgetting rule:*** $a^s \rightarrow \lambda$*, $s \in \mathbb{N}$ with $s \geq 1$, with the restriction that $a^s \notin L(E)$ for any spiking rule in $R_i$.*
- $syn \subseteq \{1, 2, \ldots, m\}^2$ *is the set of synapses, with $(i, i) \notin syn$ for $1 \leq i \leq m$.*
- *in and out are the designated input and output neurons, respectively.*

**Definition 2 (Semantics).** *The configuration of an SN P system at time $t$ is given by $C_t = \langle k_1/t_1, \ldots, k_m/t_m \rangle$, where $k_i$ denotes the number of spikes in neuron $\sigma_i$ and $t_i \geq 0$ indicates the remaining time steps until the neuron's refractory period ends (with $t_i = 0$ meaning the neuron is open). The initial configuration is $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$.*

***Rule applicability.*** *A spiking rule $E/a^c \rightarrow a^p; d$ in neuron $\sigma_i$ is enabled at time $t$ if: (i) the neuron is open ($t_i = 0$), (ii) the neuron contains $k_i$ spikes such that $a^{k_i} \in L(E)$, and (iii) $k_i \geq c$. A forgetting rule $a^s \rightarrow \lambda$ is enabled only if the neuron contains exactly $s$ spikes and no spiking rule is enabled. This holds true by definition due to the restriction $a^s \notin L(E)$ for all spiking rules in $R_i$.*

***Rule application.*** *When a spiking rule $E/a^c \rightarrow a^p; d$ fires in neuron $\sigma_i$:*

1. *$c$ spikes are consumed from $\sigma_i$;*
2. *if $d = 0$, then $p$ spikes are immediately sent to all neurons $\sigma_j$ where $(i, j) \in syn$;*
3. *if $d \geq 1$, the neuron enters a refractory period during which it cannot receive spikes or fire rules, and after $d$ time steps, $p$ spikes are sent to all post-synaptic neurons.*

*A forgetting rule $a^s \rightarrow \lambda$ simply removes $s$ spikes from the neuron without producing a spike.*

***Transition.*** *Configurations evolve in discrete time steps according to two principles [18]: (i) sequentiality at neuron level—each neuron applies at most one rule per time step, chosen non-deterministically if multiple rules are enabled; and (ii) maximal parallelism at system level—all neurons that can apply a rule*

*must do so. A transition $C_t \Rightarrow C_{t+1}$ is thus the resulting one-step, system-wide update: all selected rules fire simultaneously, and $C_{t+1}$ is obtained by, for each neuron, consuming the spikes used by its fired rule, delivering the spikes sent along incoming synapses by neurons that fired with delay 0, and advancing any pending delays and refractory periods.*

***Computation.*** *A computation is a sequence of transitions $C_0 \Rightarrow C_1 \Rightarrow \cdots$ starting from the initial configuration $C_0$. It halts if it reaches a configuration in which no rule is applicable in any neuron and all neurons are open.*

***Output interpretation.*** *Various conventions exist for reading the result of an SN P system computation [16,18]: (i) the time interval between the first two spikes of the output neuron (the classic number-generating convention); (ii) the total number of spikes emitted, or the spike count of a designated neuron when the system halts; (iii) the spike train—a binary sequence where a '1' at position $t$ indicates that the output neuron spiked at time $t$ and '0' otherwise; and (iv) for systems used as recognizers or deciders, simply whether the output neuron ever spikes (accept vs. reject). In this study we adopt convention (iv): the SFDS acts as a decider, so we interpret a spike reaching the output neuron at any step $t > 0$ as a positive verdict (i.e., a SYN flood is flagged), and the absence of any output spike throughout the computation as a negative verdict (i.e., no attack).*

### 2.4 Weighted Spiking Neural P Systems

Given this, standard SN P systems restrict spike counts to non-negative integers and gate rule firing on regular expression (RegEx) matching against those counts. However, for scenarios that require computing weighted aggregates, a more expressive variant is needed.

Wang et al. [29] thus introduced *Spiking Neural P Systems with Weights* to address this. Weighted Spiking Neural P systems (WSN P systems) allow for attaching a numerical weight to each synapse, therefore replacing the discrete spike count of each neuron with a *potential* that incoming weighted spikes either increase or decrease.

**Definition 3 (Syntax).** *Formally, a WSN P system of degree $m \geq 1$, where $m$ is the number of neurons, is defined in [29] as a construct*

$$\Pi = (\sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out),$$

*where:*

- $\sigma_i = (p_i, R_i)$ *is the $i$-th neuron, for $1 \leq i \leq m$:*
  - $p_i \in \mathbb{R}_c$ *is the initial potential stored in $\sigma_i$ (replacing the spike count $n_i$ of standard SN P systems), where $\mathbb{R}_c$ denotes the set of computable real numbers.*
  - $R_i$ *is a finite set of spiking rules of the form $T_i/d_s \rightarrow 1$, for $s = 1, 2, \ldots, n_i$ with $n_i \geq 1$, where $T_i \in \mathbb{R}_c$ with $T_i \geq 1$ is the firing threshold of $\sigma_i$, and $d_s \in \mathbb{R}_c$ with $0 < d_s \leq T_i$ is the amount of potential consumed when that rule fires. Note that the threshold $T_i$ belongs to the neuron, while the subscript $s$ indexes the rules that share it.*

- $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\} \times \mathbb{R}_c$ *is the set of weighted synapses: each triple* $(i, j, r) \in syn$ *denotes a synapse from* $\sigma_i$ *to* $\sigma_j$ *with weight* $r$*. Here* $i \neq j$*, so self-synapses are not allowed, and* $r \neq 0$*, so every synapse carries a nonzero weight. There is at most one synapse for each pair* $(i, j)$*.*
- $in, out \in \{1, 2, \ldots, m\}$ *indicate the designated input and output neurons, respectively.*

Note that Wang et al. [29] do not require these values to range over all of $\mathbb{R}$. As a matter of fact, they even prove that restricting them costs nothing in computing power. Specifically, they show that WSN P systems whose weights, thresholds, and consumed potentials are drawn from integers compute exactly the same family of sets as those drawn from either rational numbers or from computable real numbers. An exception to this equivalence, however, is the set of natural numbers, which is strictly weaker.

Given this, the SFDS that will be defined in Section 3.2 will fall under the integer case. In other words, every weight, firing threshold, and initial spike content it uses is an element of $\mathbb{Z}$. This is a restriction of the model of Wang et al., and not an extension of it.

**Definition 4 (Semantics).** *The configuration of a WSN P system at time* $t$ *is the vector of neuron potentials* $\langle p_1^{(t)}, \ldots, p_m^{(t)} \rangle$*, with initial configuration* $\langle p_1, \ldots, p_m \rangle$*.*

***Rule applicability.*** *A spiking rule* $T_i/d_s \to 1$ *in* $\sigma_i$ *is enabled at time* $t$ *if and only if the potential of* $\sigma_i$ *is exactly equal to its firing threshold, that is,* $p_i^{(t)} = T_i$*. Note that this is a strict equality: a neuron does not fire when its potential merely exceeds the threshold. Note as well that when* $p_i^{(t)} = T_i$ *holds, every rule of* $R_i$ *is enabled at once, and exactly one of them is chosen nondeterministically.*

***Rule application.*** *When the rule* $T_i/d_s \to 1$ *fires in* $\sigma_i$*, an amount* $d_s$ *of potential is consumed. Hence, the potential of* $\sigma_i$ *now becomes* $T_i - d_s$ *and one unit potential (a spike) is produced. That spike is then delivered along every synapse* $(i, j, r) \in syn$*, and each post-synaptic neuron* $\sigma_j$ *receives a quantity of potential equal to the weight* $r$*, which is added to the potential already present in* $\sigma_j$*. Note that since* $r$ *may be negative, an arriving spike can decrease a neuron's potential as well as increase it.*

***Potential that misses the threshold.*** *If the potential of* $\sigma_i$ *is strictly below its threshold, the neuron returns to the resting potential* $0$*; if it is strictly above, the potential is left unchanged. Collecting the three cases, if* $\sigma_i$ *holds potential* $p$ *and receives potential* $k$ *at step* $t$*, then at step* $t + 1$ *its potential* $p'$ *is given by*

$$p' = \begin{cases} k, & \text{if } p < T_i, \\ p - d_s + k, & \text{if } p = T_i \text{ and rule } T_i/d_s \to 1 \text{ is applied}, \\ p + k, & \text{if } p > T_i. \end{cases}$$

***Transition.*** *As with standard SN P systems (see Section 2.3), the WSN P system also observes both (i) sequentiality at neuron level, since each neuron applies at most one rule per step, and (ii) maximal parallelism at system level,*

*since every neuron whose potential equals its threshold must choose and apply a rule. Thus, a transition $C_t \Rightarrow C_{t+1}$ is the resulting one-step, system-wide update, obtained by applying the case analysis above to every neuron simultaneously. Note that the rule form $T_i/d_s \to 1$ carries no delay parameter, so WSN P systems have neither delays nor refractory periods, unlike the standard SN P systems of Definition 1.*

***Computation.*** *A computation is a sequence of transitions $C_0 \Rightarrow C_1 \Rightarrow \cdots$ starting from the initial configuration $C_0$. It halts when it reaches a configuration in which no rule can be used. As with standard SN P systems, the moments at which the output neuron emits a spike form the spike train of the computation.*

### 2.5 Spiking Neural P Systems in Cybersecurity

There have also been some prior works that have applied SN P systems to cyber-threat detection. For instance, Pleșa et al. [22] proposed a Cyber-SN P system model that was adapted from layered SN P systems and a standard neural network architecture.

They trained their system with a modified Widrow-Hoff learning algorithm. Their study applied a sigmoid function to the output and then adjusted the weights until a target training accuracy was reached.

They then evaluated the Cyber-SN P system based on three cybersecurity issues: malware detection, phishing detection, and spam detection. They also evaluated the model in terms of accuracy, precision, recall, and F1 score. Finally, they compared the results to perceptron-style Artificial Neural Network, reporting that Cyber-SN P systems achieved either comparable or slightly better metrics while requiring far fewer training epochs.

Idowu et al. [15] surveyed intrusion-detection principles, approaches, and flaws. Importantly, however, they demonstrated how a SN P system could be constructed to encode temporal and count-based intrusion signatures. This is useful to us as it is the premise of this study.

They also further advocated for the suitability of SN P systems in both known-signature detection as well as in behavior-based anomaly detection due to their parallelism, synchronization, nonlinearity, and readable graph structure.

Finally, in a different paper, Idowu et al. [14] proposed a trapezoidal Fuzzy Reasoning SN P system for DoS and SYN-flood detection. They started by first selecting key features from a dataset (i.e., `duration`, `src_bytes`, `dst_bytes`, and `count`) and represented neuron potentials and rule certainty values as trapezoidal fuzzy numbers.

Briefly, a trapezoidal fuzzy number is a number from a fuzzy set on the real line specified by four real parameters ($a \leq b \leq c \leq d$). Note that unlike classical sets, where an element either belongs to or does not belong to a set, fuzzy sets allow for degrees of membership, meaning an element can partially belong to a set with a membership value ranging between 0 and 1.

In this case, its membership function $\mu(x)$ increases linearly from 0 to 1 on the interval $[a, b]$, stays equal to 1 on $(b, c]$, then decreases linearly from 1 to 0 on $(c, d]$, and is 0 outside $[a, d]$. They are useful as they could provide an efficient

and compact way of representing either uncertain or imprecise quantities (such as rule certainty and neuron potential ranges).

To continue, they encoded fuzzy production rules that mapped combinations of the four aforementioned features to DoS likelihood levels. They then evaluated the system with a confusion matrix and reported low false negatives (0.02%) as well as false positives (0.25%), yielding an attack detection rate of 99.88% and a false alarm rate of 0.31%.

## 3 Methodology

This section develops the SFDS in full. In here, we state the detection problem formally, define its three features, define the weighted SN P system that computes them, and finally work through how the traffic is captured, encoded, and processed layer by layer.

### 3.1 The SYN Flood Detection Problem and its Features

The SFDS monitors three features to determine whether a sample of network traffic exhibits signs of a SYN flood attack. Before defining these features, we first state the detection problem formally.

**Definition 5 (Packet Type Alphabet).** *Let $\mathcal{T}$ denote the packet type alphabet, where $\mathcal{T}$ is defined as*

$$\mathcal{T} = \{\mathrm{SYN_n},\ \mathrm{SYN_a},\ \mathrm{ACK},\ \mathrm{Other}\},$$

*where* $\mathrm{SYN_n}$ *and* $\mathrm{SYN_a}$ *are normal and anomalous SYN packets respectively,* ACK *is an ACK packet, and* Other *groups all other observed packet types. An input to the SFDS is therefore a tuple $P = (P_1, \ldots, P_M)$ of $M$ concurrent per-source streams, where each $P_j = (p_{j,1}, \ldots, p_{j,n_j}) \in \mathcal{T}^*$ is the sequence observed from source $j$. Note that when a feature does not depend on the source split, we treat $P$ as the single combined sequence $(p_1, \ldots, p_n)$ of all sources' packets.*

**Definition 6 (Anomalous SYN Packet).** *An anomalous SYN packet is a SYN packet whose source IP address is spoofed, which we mark by drawing such addresses from the reserved test-net ranges of RFC 5737 [4]. Such packets are synthetically generated and labeled during traffic generation, and they correspond to the symbol $\mathrm{SYN_a} \in \mathcal{T}$ of Definition 5.*

**Definition 7 (SYN Flood Detection Problem).** *The SYN flood detection problem asks for a decision function*

$$D : (\mathcal{T}^*)^M \to \{0, 1\}$$

*such that $D(P) = 1$ whenever the input $P$ exhibits a SYN flood attack and $D(P) = 0$ otherwise. We construct $D$ from a family of feature detectors*

$$\{f_j : (\mathcal{T}^*)^M \to \{0, 1\}\}_{j \in J}$$

*combined under a voting rule* $V : \{0,1\}^{|J|} \to \{0,1\}$. *In this paper,* $J = \{\text{rate, ratio, pattern}\}$ *and* $V$ *is the 2-out-of-3 majority rule:*

$$D(P) = V\big(f_{\text{rate}}(P),\ f_{\text{ratio}}(P),\ f_{\text{pattern}}(P)\big) = \mathbb{1}\left[\sum_{j\in J} f_j(P) \geq 2\right]. \qquad (1)$$

The three aforementioned feature detectors are defined below. However, note that their default threshold values are stated for convenience of discussion: any value may be freely substituted without altering the system's design (see Section 3.12). This is because the defaults themselves are not drawn from an external standard; rather, they are derived empirically as will be discussed in Section 4.4, which identifies the range of thresholds within which each feature separates the traffic reliably.

**Definition 8 (`syn_packet_rate`).** *The* ***`syn_packet_rate`*** *feature measures the velocity of SYN packets: the number of SYN packets (normal or anomalous) that arrive within a fixed-width time window, aggregated across all sources. Formally, for a window* $P$*,*

$$f_{\text{rate}}(P) = \mathbb{1}\left[\left|\{i : p_i \in \{\text{SYN}_\text{n}, \text{SYN}_\text{a}\}\}\right| \geq K\right],$$

*where* $K \in \mathbb{Z}_{\geq 1}$ *is the rate threshold (set to* $K = 60$ *by default).*

**Definition 9 (`syn_ack_ratio`).** *The* ***`syn_ack_ratio`*** *feature measures, per source, how large a share of a source's handshake-relevant packets are SYNs rather than ACKs. Specifically, for each per-source stream* $P_j$ *of the window, let*

$$S_j = \left|\{k : p_{j,k} \in \{\text{SYN}_\text{n}, \text{SYN}_\text{a}\}\}\right|, \qquad A_j = \left|\{k : p_{j,k} = \text{ACK}\}\right|$$

*be its SYN and ACK counts. The feature triggers if any single source's SYN-to-ACK proportion exceeds the threshold:*

$$f_{\text{ratio}}(P) = \mathbb{1}\left[\max_{\substack{1\leq j\leq M \\ S_j+A_j>0}} \frac{S_j}{S_j + A_j} > R\right],$$

*where* $R \in [0,1]$ *is the ratio threshold (set to* $R = 0.70$ *by default).*

**Definition 10 (`anomalous_syn_pattern`).** *The* ***`anomalous_syn_pattern`*** *feature counts the number of anomalous SYN packets observed in the input, regardless of order:*

$$f_{\text{pattern}}(P) = \mathbb{1}\left[\left|\{i : p_i = \text{SYN}_\text{a}\}\right| \geq \theta_{\text{pattern}}\right],$$

*where* $\theta_{\text{pattern}} \in \mathbb{Z}_{\geq 1}$ *is the pattern threshold (set to* $\theta_{\text{pattern}} = 3$ *by default).*

### 3.2 The Weighted Spiking Neural P SYN Flood Detection System

To solve the SYN flood detection problem, we constructed a variation of the standard SN P system that utilizes weighted synapses. The Weighted SN P SFDS is therefore defined as follows:

**Definition 11.** *Formally, the* ***Weighted SN P SFDS*** *of degree* $6M+8$*, where* $M \geq 1$ *is the number of concurrent input streams, is a tuple*

$$\Pi_{SFDS} = (\{a\}, \sigma_1, \sigma_2, \ldots, \sigma_{6M+8}, syn, In, out),$$

*where:*

- $\{a\}$ *is the singleton alphabet of Definition 1, where* $a$ *denotes a spike.*
- $\sigma_i = (n_i, R_i)$ *is the* $i$*-th neuron, for* $1 \leq i \leq 6M + 8$*, where* $n_i \in \mathbb{Z}_{\geq 0}$ *is its initial spike content and* $R_i$ *is a finite set of rules of the following three forms:*
  1. ***Spiking rule:*** $E/a^c \to a^p; d$*, exactly as in Definition 1, which emits* $p$ *spikes after delay* $d$*.*
  2. ***Forgetting rule:*** $a^s \to \lambda$*, exactly as in Definition 1, which discards* $s$ *spikes without emitting any.*
  3. ***White hole rule:*** $a^k(a)^*/a^k(a)^* \to a; 0$ *with* $k \geq 1$*, which becomes enabled once the neuron holds at least* $k$ *spikes and, upon firing, consumes its entire content and emits a single spike [3].*
- $syn \subseteq \{1, \ldots, 6M + 8\} \times \{1, \ldots, 6M + 8\} \times (\mathbb{Z} \setminus \{0\})$ *is the set of weighted synapses, where each* $(i, j, w) \in syn$ *denotes a synapse from* $\sigma_i$ *to* $\sigma_j$ *with integer weight* $w$*. As in Definition 3,* $i \neq j$ *and* $w \neq 0$*, and there is at most one synapse for each pair* $(i, j)$*.*
- $In = \{In_1, \ldots, In_M\}$ *is the set of designated input neurons, one per source, and out is the single designated output neuron.*

*The input to* $\Pi_{SFDS}$ *is the tuple* $P = (P_1, \ldots, P_M)$ *of per-source streams of Definition 5, where each* $P_j$ *holds at most* $H$ *packets and is terminated by an end-of-stream marker. Stream* $P_j$ *is fed to input neuron* $In_j$ *at one packet per time step, and each packet enters as the number of spikes assigned to its type by the encoding scheme that will be defined in Section 3.5.*

Following this, however, there are a few points that we would like to discuss further below.

First, note that this definition fixes only the shape of the system. The initial spike content $n_i$ and the rule set $R_i$ of each individual neuron, together with the weight of each synapse, are specified layer by layer in Sections 3.6–3.11. Moreover, just like the feature thresholds, $M = 3$ and $H = 600$ are defaults for convenience of discussion. At $M = 3$, the system therefore has $6M + 8 = 26$ neurons.

Second, note that the white hole rule is a convenience rather than a necessity. To demonstrate, recall first that a neuron's spike content within a window is bounded, say by $B$. This therefore means that the rule $a^k(a)^*/a^k(a)^* \to a; 0$ can

be simply expanded into a list of standard rules $a^i/a^i \to a;0$ for every content $i$ from $k$ up to $B$. Since there are only $B-k+1$ such values, the resulting rule set is thus finite and behaves identically.

Third, note that the standard model of Definition 1 would not have been directly suitable for this study. This is because of two fundamental restrictions: first, that a spike only ever adds to the receiving neuron's content, so no neuron can decrement another, and second, that a firing neuron sends the same number of spikes to all of its targets at once. The SFDS, however, needs both restrictions lifted. This is because its ratio detector, which will be constructed in Section 3.9, is a signed accumulator that adds on every SYN packet and subtracts on every ACK packet. Its SYN filters, meanwhile, must deliver both (a) a large increment to that detector and (b) a single spike to the shared rate and pattern detectors in the same step. Fortunately, weighted synapses allow us to do both since not only does each synapse carry its own weight but these very weights may also be negative. Of course, since standard SN P systems are computationally universal [16], the claim is **not** that no unweighted system could decide the same inputs. Rather, the claim is that since no standard rule lets one neuron decrement another, a weight-free SFDS would need a different ratio detector—one whose constants could no longer be derived from the synapse weights.

Finally, note that this model also differs from the pure weighted SN P system of Wang et al. [29]. This is because although the synapses are now weighted, the firing rules still retain the standard regex-based form of $E/a^c \to a^p;d$ rather than the threshold-equality form of $T_i/d_s \to 1$. To further elaborate, first recall from Definition 3 that a WSN P system neuron (a) discards any potential below its threshold and (b) fires only when its potential exactly equals the corresponding threshold. The ratio detector of Section 3.9, however, needs the opposite behavior on both counts: to accumulate a signed total while below its threshold as well as to fire when that total *crosses* the threshold rather than only when it lands exactly on it. Therefore, retaining regex-based rules over weighted synapses is what makes the construction expressible at all.

### 3.3 Traffic Generation and Capture

To generate the network traffic, one would usually use a *physical*, distributed network setup: a single victim server receiving traffic from $M$ distinct client devices. However, we opted to emulate this topology on *one* Linux host using virtual *network namespaces* instead. This is because real network infrastructure may silently drop spoofed-source packets in transit [10], whereas the local software bridge sends every packet reliably and counts it exactly once at the victim.

Crucially, each attacker namespace is also assigned a distinct, deterministic hardware (MAC) address. This is because, for our threat model, a flood can spoof its source *IP* but **not** its layer-2 *MAC*, so the victim-side capture can still attribute every frame to the `device` that sent it.

Each attacker runs a `Scapy` [24] traffic generator whose parameters control its stream: the packet rate, the fraction of handshakes completed, and whether source IPs are spoofed from the RFC 5737 ranges. The victim, in turn, runs a

minimal TCP server that keeps a port open so that the kernel can answer each SYN with a SYN-ACK. The victim also runs, in parallel, a capture script that records every packet to a CSV file.

### 3.4 Input Windowing

As they currently are, our newfound captures are *arbitrarily long* packet logs. Therefore, we must first segment them into windows. However, the manner in which we do so directly affects the correctness of the system.

For instance, suppose that we naïvely cut the capture at fixed time boundaries. The first problem with this is that a sudden burst of traffic can pack an unbounded number of packets into a single window. The second problem is that a boundary can fall right between a SYN packet and its corresponding ACK packet. The SYN would then be stranded without its acknowledgment, making the ratio feature report a half-open connection when, in fact, the handshake was completed. This would yield a false positive caused by the cutoff system employed rather than by the network traffic itself.

Thus, to prevent both problems, we preprocess the input using fixed-width time windows of $\Delta t$ (1 second by default) alongside two safeguards. First, we allow for a brief *grace* period (0.05 s by default) so that in-flight handshakes can finish, thereby keeping a SYN and its trailing ACK within the same window. Second, we also fix a *hard limit* of $H$ packets (600 by default) that forces a window to close regardless.

Note that it is this fixed $\Delta t$ that makes each window's SYN count a measure of *velocity* (as required by the rate feature of Section 3.1), while $H$ is the very same per-source packet count limit that bounds the ratio detector's arithmetic in Section 3.2.

Each window is then split by `device` into its per-source substreams. In other words, the packets from attacker $i$ form the input stream for the input neuron $In_i$, and so on. Note as well that only the attacker-to-server packets are encoded; the victim's replies are excluded so that the victim is never mistaken for a source.

### 3.5 Packet Encoding

The SN P SFDS then employs a spike-based encoding scheme to convert packet types into spike counts. Each packet type is mapped to a specific number of spikes: normal SYN packets are encoded as $a^3$ (3 spikes), anomalous SYN packets as $a^2$ (2 spikes), ACK packets as $a^4$ (4 spikes), stream terminators as $a^5$ (5 spikes), and all other packet types as $a^1$ (1 spike).

### 3.6 System Architecture and Input Layer

The SN P SFDS is organized into five layers — input, filtration, detection, flagging, and output — each performing a specific function in the detection pipeline. We describe each below, with the full Websnapse diagram shown in Figure 2.

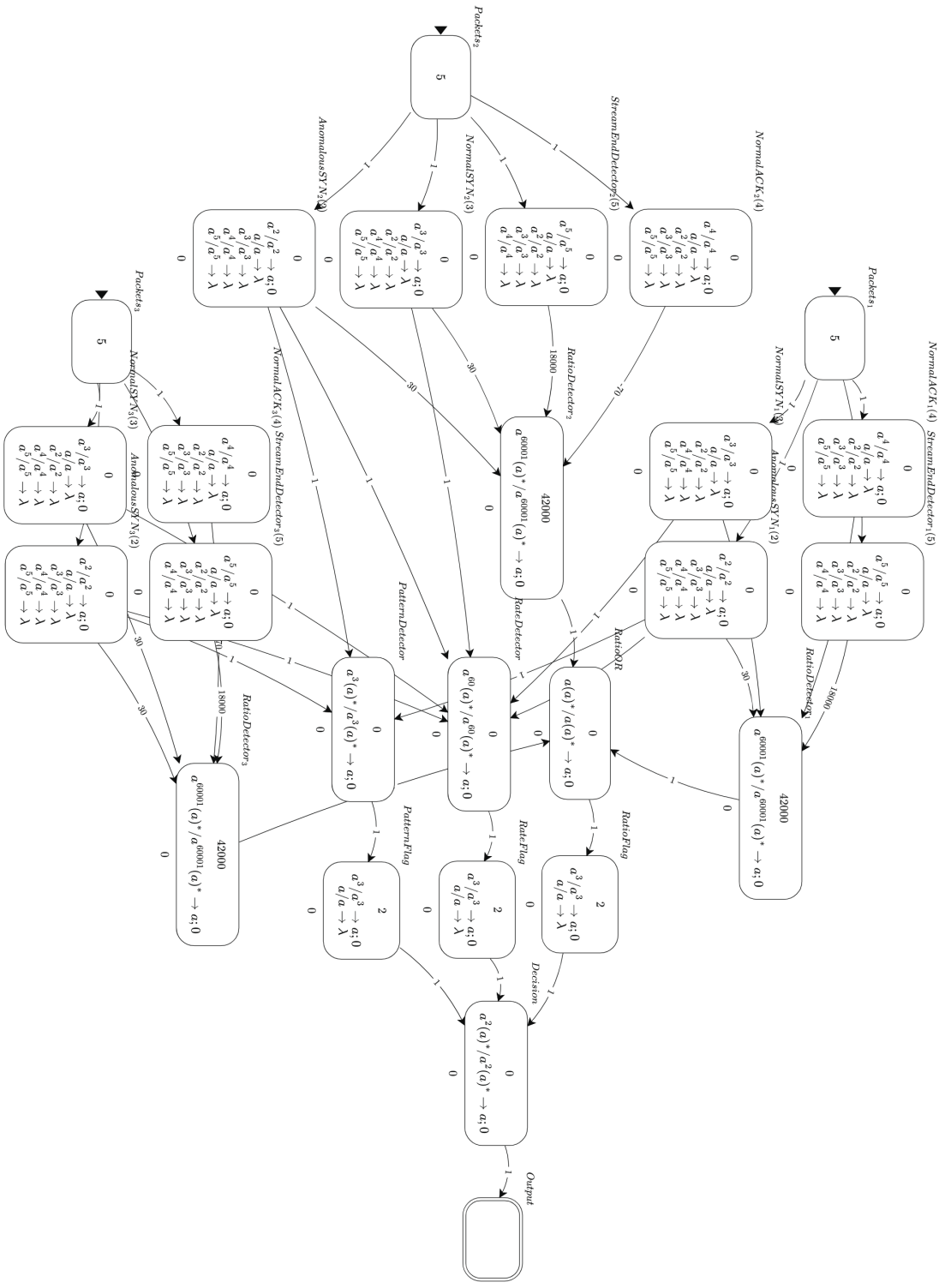



Fig. 2: Websnapse Diagram of the Weighted SN P SFDS

Each input source has its own dedicated chain of six neurons: one input neuron (the **Input Layer**), three filter neurons (`AnomalousSYN_i`, `NormalSYN_i`, `NormalACK_i`), one stream-end detector, and one ratio detector.

The remaining eight neurons are then shared globally: three feature detectors (`RateDetector`, `RatioOR`, and `PatternDetector`), three flag neurons (`RateFlag`, `RatioFlag`, `PatternFlag`), one decision neuron, and one output neuron. In total, the system has $6M+8 = 26$ neurons at $M = 3$, where $M$ is the number of streams from Definition 5.

Note that the per-source replication allows us to independently process each input stream, while the shared aggregators allow us to handle the features whose semantics span all sources (such as the global SYN velocity and the total anomalous-SYN count, as explained below).

### 3.7 Filtration Layer

The **Filtration Layer** routes each packet type to the detectors that care about it. As mentioned, each source $i$ has four filter neurons, one per meaningful symbol: `AnomalousSYN_i(2)`, `NormalSYN_i(3)`, `NormalACK_i(4)`, and `StreamEndDetector_i(5)`. Since each is fed by a single source's stream (with a synapse weight of 1), it sees one symbol per clock tick. Therefore, these filter neurons fire on an exact match to their own symbol via $a^s/a^s \to a; 0$ and discard every other symbol with a forgetting rule $a^{s'} \to \lambda$.

Given this, the outgoing synapses of these filter neurons connect to the three features as follows:

- Each SYN filter (`AnomalousSYN_i` and `NormalSYN_i`) sends a single spike to the shared `RateDetector` (weight $+1$), and a weight of $+30$ to its own `RatioDetector_i`.
- The ACK filter (`NormalACK_i`) sends a weight of $-70$ to its `RatioDetector_i`.
- The anomalous-SYN filter (`AnomalousSYN_i`) additionally sends a single spike to the shared `PatternDetector` (weight $+1$).

Note that it is these very $+30$ and $-70$ weights that allow `RatioDetector_i` to evaluate the per-source SYN:ACK proportion, as will be explained in Section 3.9.

**Stream-End Neurons.** Since we accept variable-length input streams, a ratio detector cannot know in advance how many packets its source will send, and its verdict is only meaningful once the whole stream has been consumed. Each source, therefore, has a `StreamEndDetector_i(5)` neuron that fires exactly once, on the end-of-stream marker $a^5$ (rule $a^5/a^5 \to a; 0$), discarding every other symbol via forgetting rules.

When `StreamEndDetector_i(5)` fires, its spike is delivered to that source's own `RatioDetector_i` along a synapse of finalizing weight 18000 (see Section 3.9), which completes the per-source ratio computation. Note that a source with no traffic is loaded with the lone marker `"5"`, so its detector fires immediately and contributes nothing.

### 3.8 Detection Layer: The Rate Feature

The core of the rate feature lies with the shared `RateDetector` neuron. It begins empty and then receives one spike from every SYN filter neuron of every source. This is so that its content becomes the running total of SYN packets across all sources in the current window. It carries the single *white hole* rule (i.e., a rule that consumes the neuron's entire content when it fires [3]; see also Section 4)

$$a^K(a)^*/a^K(a)^* \to a; 0,$$

which fires the moment that total reaches the threshold $K$ and then resets. Since every window spans the same fixed interval $\Delta t$, this per-window SYN count is thus a direct measure of SYN *velocity*.

### 3.9 Detection Layer: The Ratio Feature

Unlike the rate and pattern features (which are aggregated globally), the `syn_ack_ratio` feature is detected *per-source*. This is because the SYN:ACK ratio is a per-handshake property: each TCP handshake involves exactly one client (the source) and the server [9]. As a result, detection happens within each source's own packet stream rather than across the merged multi-source input. Aggregation across all sources is then handled *after* the per-source detection through a single shared OR-gate. The ratio feature is therefore implemented with the use of two neurons: `RatioDetector_i` (per-source, replicated $M = 3$ times) and the shared `RatioOR`.

However, the difficulty behind this feature is that a proportion is a division, which a spike count cannot express directly. Moreover, as with the rate, the packet count is not known in advance. We therefore have to encode the ratio proportion algebraically instead, through a construction that must meet these three requirements: it must compute the right quantity, it must never go negative, and it must fire exactly when the proportion is exceeded and not before. We explain our solution to each below:

**Algebraic Encoding of the SYN:ACK Proportion.** For a single source with $S$ SYN and $A$ ACK packets, the condition we wish to detect is $\frac{S}{S+A} > R$ as per Definition 9, where the ratio threshold is written as a fraction $R = r/b$ (default $r = 70$, $b = 100$, i.e. $R = 0.70$). Multiplying through by the positive quantity $b(S + A)$ and rearranging gives the equivalent integer condition

$$(b - r)S - rA > 0. \tag{2}$$

Accordingly, each `RatioDetector_i` accumulates $+(b - r)$ per SYN and $-r$ per ACK: the $+30$ and $-70$ weights of Section 3.7. Thus after an input stream, its accumulated contribution is exactly $(b - r)S - rA$.

**Non-Negativity and the Initial Content $C$.** Recall that SN P systems require all neuron contents to remain non-negative throughout any computation (i.e., $n_i \geq 0$ as per [16]). With the per-packet contributions established above, however, every ACK packet contributes $-r$ to the spike content of `RatioDetector_i`. If the detector were to start at zero, even a single ACK packet would cause its content to become negative.

To prevent this, we identify the worst-case input scenario for non-negativity: the stream in which every packet is an ACK. Since a source can send at most $H$ packets per window (Section 3.2), the contribution under this scenario is exactly $-rH$.

Therefore, we must choose a value $C$ (where $C$ is the initial spike content of `RatioDetector_i`) that is large enough such that the content remains non-negative even under the worst case. In other words, we require

$$C - rH \geq 0.$$

Given this, the smallest value of $C$ that satisfies this constraint is

$$C = rH. \tag{3}$$

Thus, for our default configuration with $r = 70$ and $H = 600$, this gives $C = 70 \cdot 600 = 42000$.

**Finalization $D$ and Threshold $T$.** Now that we have an equation for the initial content $C$, what remains is to determine both the finalizing weight $D$ (the weight on the synapse from `StreamEndDetector_i(5)` to `RatioDetector_i`) and the firing threshold $T$ (the spike count at which `RatioDetector_i` fires its rule). We derive these by working through the three constraints below.

The first constraint is that `RatioDetector_i` must *not* fire prematurely — that is, before its source's stream has ended. The worst-case scenario for premature firing is the stream in which every packet is a SYN, since each contributes $+(b - r)$. After $H$ such packets, the content is

$$C + (b - r)H = rH + (b - r)H = bH.$$

Therefore, to prevent firing at any point mid-stream, the threshold must satisfy $bH < T$.

The second constraint is that `RatioDetector_i` *must* fire whenever the proportion condition $(b - r)S - rA > 0$ of Equation (2) has been satisfied. After `StreamEndDetector_i(5)` has delivered its $D$-weighted spike, the content is $C + (b - r)S - rA + D$. Since $S$ and $A$ are integers, the smallest satisfying case is $(b - r)S - rA = 1$, and we therefore require

$$C + 1 + D \geq T.$$

Lastly, the third constraint is the converse: `RatioDetector_i` must *not* fire whenever $(b-r)S-rA \leq 0$. The maximum such case is $(b-r)S-rA = 0$, under which the content would be $C + D$. Thus, we require

$$C + D < T.$$

Combining the second and third constraints, the smallest threshold value that satisfies both is $T = C + D + 1$; this trailing $+1$ is precisely what makes the detected inequality strict. Substituting $T$ into the first constraint and simplifying then gives $(b-r)H \leq D$, so the smallest valid finalizing weight is $D = (b-r)H$. Together:

$$D = (b-r)H, \qquad T = C + D + 1 = bH + 1. \tag{4}$$

For our default configuration ($r = 70$, $b = 100$, $H = 600$), this gives $D = 18000$ and $T = 60001$, realized by the rule $a^T(a)^*/a^T(a)^* \to a; 0$.

**The `RatioOR` Neuron.** Finally, `RatioOR` combines the per-source ratio outputs into a single global verdict. It begins empty, receives one spike from each `RatioDetector_i` that fired, and carries the single rule $a(a)^*/a(a)^* \to a; 0$, firing if and only if *at least one* source exceeded the threshold. Its output then gets sent to the flagging layer as will be described in Section 3.11.

### 3.10 Detection Layer: The Pattern Feature

Lastly, we have the `anomalous_syn_pattern` feature, which is also a *global* aggregate. It is implemented through the single shared `PatternDetector` neuron.

It begins with an empty spike content, then accumulates one spike per anomalous SYN packet from each `AnomalousSYN_i(2)` filter, and then fires once its content reaches the firing threshold. Its output then propagates to the flagging layer that follows after.

By default, it consists of the sole rule $a^3(a)^*/a^3(a)^* \to a; 0$, which fires whenever the content reaches at least 3. This value, however, is adjustable as we shall see in Section 4.4.

### 3.11 Flagging and Output Layers

The role of the **Flagging and Output Layers** is to implement the *2-out-of-3 majority voting scheme*. In other words, they determine whether at least two of the three feature detectors have fired to produce the final output.

**Design Rationale.** Before proceeding, however, note that there are three constraints that must be satisfied:

1. Each feature only needs to be flagged at least *once* within the entire spike train input window.
2. Once a feature has been flagged, it should *never* be unflagged by subsequent packets within the same spike train input window.
3. A single feature triggered *multiple times* should **not** be counted as multiple features.

**The Flag Neurons.** To satisfy these constraints, each feature detector is connected to its own *flag neuron* (`RateFlag`, `RatioFlag`, and `PatternFlag`), which collapses any number of firings of its detector into a single spike as explained below:

1. First, each flag neuron initially starts with 2 spikes.
2. Then, the *first* time its detector fires, the flag neuron receives 1 spike and reaches exactly 3. This triggers its $a^3/a^3 \to a; 0$ rule: it sends a single spike to the `Decision` neuron and consumes all 3 spikes, dropping to 0.
3. Afterwards, every *later* firing of the same detector therefore finds the flag neuron at 0 spikes, where the arriving spike matches only the forgetting rule $a \to \lambda$ and is discarded without output.

Hence, no matter how many times a detector fires, exactly one spike per feature ever reaches the vote.

**The Decision and Output Neurons.** The `Decision` neuron then implements the vote itself. It begins with an empty initial spike count, receives the spikes of the flag neurons, and then carries the single white hole rule

$$a^2(a)^*/a^2(a)^* \to a; 0,$$

which fires once its content reaches 2 (i.e., exactly when at least two of the three features have been flagged). No single feature can therefore raise the alarm alone. This is deliberate: a lone feature trip at low volume (e.g., a client retrying an unresponsive service, which trips just the ratio feature) is ambiguous on its own, so requiring corroboration could aid in reducing such false positives.

Finally, when `Decision` fires, its spike is delivered to the designated `Output` neuron. Following the decider convention adopted in Section 2.3, this spike constitutes the system's positive verdict: a SYN flood is flagged for the current window.

### 3.12 Modifying Threshold Values

Modifying the threshold values before running a simulation can be done either manually or through code.

**Manual Changes.** Manually changing the threshold values of the detector neurons can be tedious, but it gets the job done. To do so, we simply look for the detector neurons in the `.json` file of the SFDS and change their respective rules (and initial spike contents) according to the calculated threshold values.

**Code Implementation.** Alternatively, the system can be constructed programmatically. Our implementation takes the parameters ($M$; $K$; $r$ and $b$; $\theta_{\text{pattern}}$; and $H$) as arguments and generates every neuron, rule, and initial spike content from the formulas of this section. In other words, changing a threshold is a single argument change, and the resulting system still implements the features exactly as defined in Section 3.1.

### 3.13 Websnapse

**Websnapse** is the tool that we used to build and visually simulate the SFDS. It is a web-based tool for simulating visual experiments on SN P systems, as well as their other variants [11,31]. The main features that the SFDS made use of include:

- **Adding** the unique nodes and edges that make up the SFDS;
- **Editing** the properties of the nodes that drive the simulation;
- **Deleting** unnecessary nodes that have no use or may be problematic for the SFDS;
- Browsing the **history** of a simulated run to analyze the results; and
- Running the simulation either **continuously** or **tick-by-tick**.

Together, these features helped us run guided and visualized simulations of the SFDS.

## 4 Results and Discussion

This section evaluates the weighted SN P SFDS across the eight traffic scenarios that will be seen in Section 4.1. We first describe the scenarios, then verify the detector against them at its default thresholds, study how each feature's threshold may be calibrated, and finally compare the system with existing detectors.

Note that while the SFDS was defined and visualized in Websnapse, we simulated our experiments in our simulator that supports white hole rules in Definition 11 found in [1]; we then validated the results of our simulator that will be seen in Sections 4.3 and 4.4 using WebSnapse, see for instance the transformation of the standard SN P (see Definition 1) and white hole rules (see Section 3.2).

### 4.1 Experiment Scenarios

Each scenario is a set of concurrent per-source streams, where a stream is characterized by four (4) configurable options: its packet rate (`pps`), its `duration`, the fraction of its connections that complete the TCP three-way handshake (`complete_handshake_ratio`), and whether its source IPs are spoofed from the RFC 5737 ranges (`spoof`). By varying these options across the three attacker devices, we can compose eight scenarios that together exercise *almost* every combination of the three features, as summarized in Table 1.

Note that every scenario keeps benign, handshake-completing background traffic running on its non-attacking devices; the feature-triggering streams are then overlaid on top of this background. In `pattern_only`, for instance, the spoofed SYNs are overlaid onto a device that also completes ordinary handshakes, so that its own ACKs keep its SYN:ACK ratio below the threshold and only the pattern feature fires.

Meanwhile, although `ratio_only` and `stealth` share the same expected label, they serve different purposes: the former isolates the ratio feature, while

Table 1: The Eight Traffic Scenarios and the Features Each is Designed to Trigger

| Scenario | Volume | Handshakes | Spoofed | Rate | Ratio | Pattern |
|---|---|---|---|---|---|---|
| `normal` | low | completed | no | F | F | F |
| `rate_only` | high | completed | no | T | F | F |
| `ratio_only` | low | one half-open | no | F | T | F |
| `pattern_only` | low | completed | yes | F | F | T |
| `rate_ratio` | high | one half-open | no | T | T | F |
| `ratio_pattern` | low | one half-open | yes | F | T | T |
| `full_attack` | high | half-open | yes | T | T | T |
| `stealth` | low | one half-open | no | F | T | F |

the latter frames it as an evasion attempt — a quieter half-open flood buried in busier background traffic — that stays under the aggregate rate threshold and is caught only by the *per-source* ratio.

Note as well that there is no `rate_pattern` scenario. This is because a spoofed SYN can never complete its handshake, as its SYN-ACK is addressed to an unreachable host. Therefore, any spoofed stream voluminous enough to drive the aggregate rate would drag its own source's SYN:ACK proportion past the threshold along the way. In other words, at large enough volumes, pattern *implies* ratio; the high-rate spoofed flood thus appears as `full_attack` rather than as any two-feature scenario.

### 4.2 Computational Complexity

Recall from Section 3.2 that the weighted SN P SFDS contains a total number of twenty-six (26) neurons, with eighteen (18) of its neurons are due to the three (3) concurrent sources for $M = 3$. This means that a weighted SN P SFDS with $M$ concurrent sources has a total number of $6M + 8$ neurons, giving $O(M)$ space complexity.

In terms of its time complexity, since the neurons run concurrently and in a feed-forward manner, we consider only the worst runtime of the longest spike train as input. The longest runtime in the feature is the RatioDetector, due to its nature of waiting until the Stream End (5) packet is received before firing. This makes the runtime $H + 6$ giving a time complexity of $O(H)$, where H is the maximum number of packets (see Section 3.2).

### 4.3 Initial Thresholds

We first run the detector at its default thresholds: $K = 60$ for the rate, $R = 0.70$ for the ratio, and $\theta_{\text{pattern}} = 3$ for the pattern. Table 2 reports, for each scenario, the ground-truth feature label of Table 1 against the label the detector produced, together with the combined alarm.

We can see that the detector reproduced the ground-truth label for all eight scenarios. The alarm also followed the 2-out-of-3 rule exactly: it fired for the

Table 2: Detector Output versus Ground Truth at the Default Thresholds

| Scenario | Expected (rate, ratio, pattern) | Observed | Alarm |
|---|---|---|---|
| `normal` | (F, F, F) | (F, F, F) | F |
| `rate_only` | (T, F, F) | (T, F, F) | F |
| `ratio_only` | (F, T, F) | (F, T, F) | F |
| `pattern_only` | (F, F, T) | (F, F, T) | F |
| `rate_ratio` | (T, T, F) | (T, T, F) | T |
| `ratio_pattern` | (F, T, T) | (F, T, T) | T |
| `full_attack` | (T, T, T) | (T, T, T) | T |
| `stealth` | (F, T, F) | (F, T, F) | F |

three scenarios in which at least two features are present and stayed silent otherwise. However, we are careful about what this shows. Since the thresholds are calibrated on this same synthetic set (see Section 4.4), the result from Table 2 demonstrates *feature separability* rather than an estimate of field detection accuracy. Such a generalization would require evaluation on external traffic, which we identify as future work (see Section 5). Note, though, that the captures themselves are genuine TCP exchanges with a live network stack (as per Section 3); *synthetic* merely refers to the scripted scenarios themselves.

### 4.4 Optimizing Thresholds

Having fixed the defaults, we can now study how each feature behaves as its threshold varies, both to justify the defaults and to answer how the thresholds may be chosen systematically rather than by hand. Using the same eight scenarios, we sweep each feature's threshold across a plausible range and record the accuracy, recall, precision, false-positive rate, and F1 at each value.

**Rate Feature.** Figure 3 sweeps the velocity threshold $K$ from 20 to 100 SYN packets per window. It shows that the detector separates all eight scenarios perfectly (accuracy, precision, recall, and F1 all 1.0, with zero false positives) across the wide band $K \in [41, 94]$. Below this band the threshold is low enough that busy-but-legitimate traffic is misread as a flood (false positives), while above it genuine floods begin to slip under the bar (false negatives).

**Ratio Feature.** Figure 4 sweeps the ratio threshold as a percentage from 50 to 90. It shows that detection is perfect across the range $[66, 74]$. Below this band we get sources with a modest but legitimate SYN:ACK imbalance being flagged, while above it a genuinely half-open source can fall just short.

**Pattern Feature.** Finally, Figure 5 sweeps the anomalous-SYN count $\theta_{\text{pattern}}$ from 1 to 10. The feature is perfect for $\theta_{\text{pattern}} \in [1, 7]$ and only begins to miss the lightly-spoofed scenarios beyond $\theta_{\text{pattern}} = 7$.

Fig. 3: Results of the Rate Detector with the 8 Scenarios of Table 1 Across the Threshold Range of 20–100

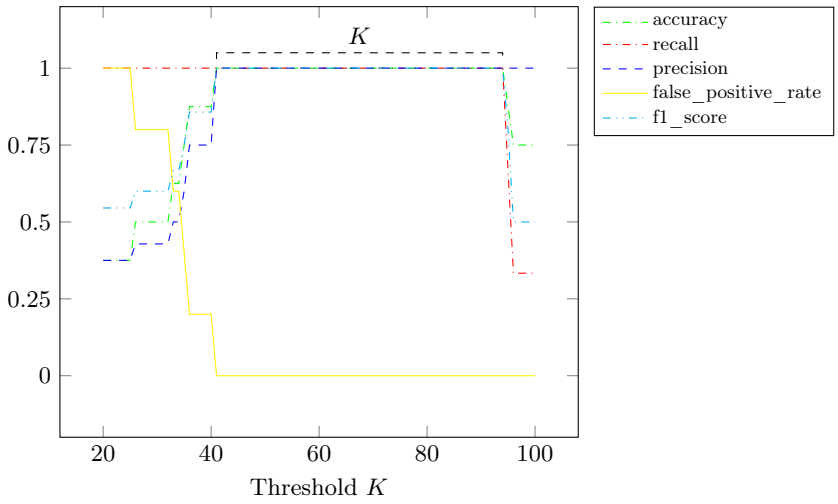



Fig. 4: Results of the Ratio Detector with the 8 Scenarios of Table 1 Across the Threshold Range of 50–90

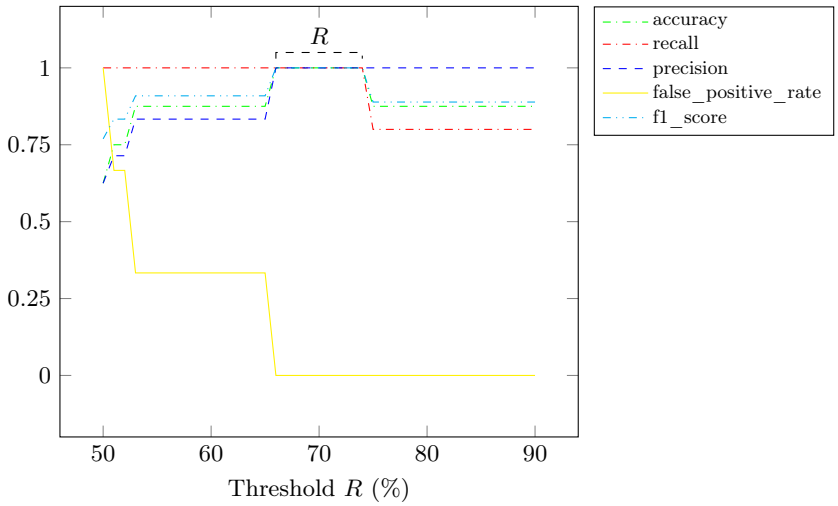
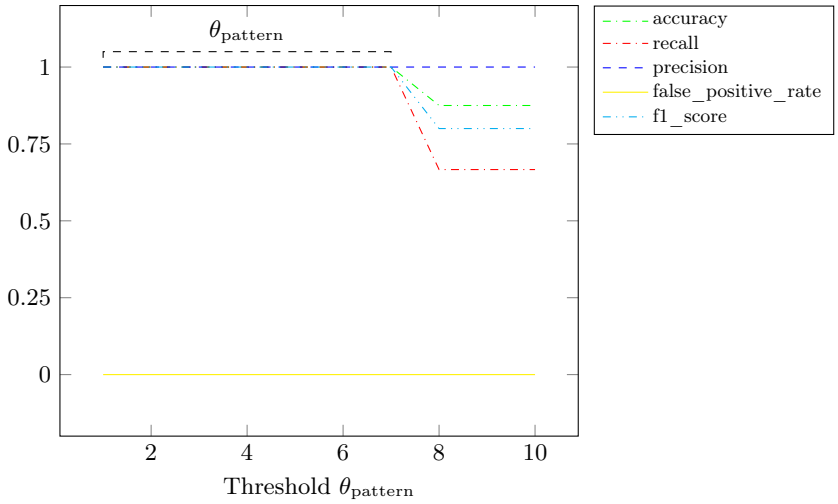


Fig. 5: Results of the Pattern Detector with the 8 Scenarios of Table 1 Across the Threshold Range of 1–10




Therefore, based on all three of these cases, the defaults of Section 4.3 ($K = 60$, $R = 0.70$, and $\theta_{\text{pattern}} = 3$) sit well within their respective bands.

### 4.5 Comparison with Existing Detection Systems

Finally, we situate the SFDS among the detection systems reviewed in Section 2.2. Note that a direct accuracy comparison would be misleading here as the reviewed systems were evaluated on different tasks, data, and metrics. For instance, Han et al. [12] report a 98% detection rate for collaborative detection on their own testbed, while Idowu et al. [14] report a 99.88% detection rate over benchmark dataset features. However, as established in Section 4.3, our own result is a separability figure on calibrated synthetic traffic rather than a field accuracy. We therefore compare the systems on their *capabilities*, along the axes that this paper targets. Table 3 summarizes this comparison, where a "–" indicates that the property is not specified by the cited work.

Given this, a few observations stand out from Table 3. First, the automaton-based detectors share our formal grounding and streaming operation, but each fixes its detection logic: the decision tree of Luckner [19] would require retraining to change behavior, while the TDFA of AlDossary et al. [2] relies on a hand-set timing constant. In contrast, every threshold of the SFDS is an explicit parameter whose rule constants are derived algebraically. Second, only the collaborative scheme of Han et al. [12] and the SFDS aggregate signals from multiple sources, and the SFDS does so *within* a single formal model rather than through a separate coordination protocol.

Table 3: Capability Comparison Between the Proposed SFDS and the Reviewed Detectors

| Capability | DT+DFA [19] | TDFA [2] | CDD [12] | Fuzzy SN P [14] | This work |
|---|---|---|---|---|---|
| Formal computational model | ✓ | ✓ | × | ✓ | ✓ |
| Training-free | × | ✓ | – | ✓ | ✓ |
| Streaming, variable-length input | ✓ | ✓ | ✓ | × | ✓ |
| Multi-source aggregation | × | × | ✓ | × | ✓ |
| Systematically calibrated thresholds | × | × | – | × | ✓ |

## 5 Conclusion and Recommendations

The present work provides a weighted SN P SFDS whose contributions, as stated in Section 1, were delivered through the following:

1. **Variable-Length Input:** An end-of-stream marker (the encoded symbol $a^5$) and per-source `StreamEndDetector_i(5)` neurons were introduced to signal when each source's stream has terminated. The per-source SYN:ACK condition was then expressed algebraically as $(b - r)S - rA > 0$ and implemented via weighted synapses (+30 per SYN and −70 per ACK at the defaults), with the initial content and finalizing weight chosen so that the verdict is computed exactly at stream end — regardless of stream length — while every spike count remains non-negative.
2. **Multi-Source Aggregation:** Per-source neuron chains were replicated for each of the $M$ input sources, with the ratio feature evaluated locally (per source, then combined through an OR-gate) and the rate and pattern features aggregated globally through shared detector neurons. A 2-out-of-3 majority vote then combines the three features into a single alarm.
3. **Systematic Thresholds:** Each feature's threshold was swept across its plausible range against the eight labeled traffic scenarios. All three features separated their target behaviors perfectly over wide threshold bands (rate $[41, 94]$, ratio $[66, 74]$, and pattern $[1, 7]$).

From the aforementioned contributions, we provide the following directions for future work, also to address some of the shortcomings of our present work:

1. **Multi-Output Verdicts:** Extend the system to be multi-output as well. Per-source outputs would identify which specific source(s) appear to be flooding, enabling targeted mitigation responses such as rate-limiting or IP blocking. Per-feature outputs, meanwhile, would surface which feature(s) triggered the detection, providing diagnostic information about the attack's nature.
2. **Validation on External Datasets:** Evaluate the SFDS on external DDoS datasets (e.g., CIC-DDoS2019) to corroborate the current results. This is so that the results can also establish actual field detection accuracy rather than just feature separability.

3. **Extension to Other Attack Vectors:** Adapt the methodology to other DDoS vectors such as UDP floods and HTTP floods. This may require new features and a corresponding extension of the encoding scheme.
4. **Alternative SN P System Variants:** Investigate alternative variants of SN P systems such as *numerical SN P systems* [30], *asynchronous SN P systems* [7], *SN P systems with learning functions* [26], and more recent ones such as wireless SN P systems [21], among others.
5. **Further verification and validation:** Verify the correctness of the simulator in the present work from [1]. It is also worth investigating the experimental validations of the present simulator with respect to other simulators for SN P systems, such as WebSnapse, P-Lingua, simulators in GPUs [31].

**Acknowledgements** F. G. C. Cabarle is supported by the 2024–2026 Scientific Productivity System of the University of the Philippines (UP), the San Francisco and Mely and Rick Rey Foundation Professorial Chair of the UP Alumni Association, and RLC 2025–2026 from the UP Diliman OVCRD.

## References

1. Abucay, J.J.D., Odhuno, S.I., Buño, K.C., Cabarle, F.G.C.: A Weighted Spiking Neural P Systems Approach to Distributed SYN Flood Detection Code Implementation. https://github.com/ShaneOdhuno05/wsnp-sfds (2026), archived at https://doi.org/10.5281/zenodo.21216153
2. AlDossary, N., AlQahtani, S., Alzaher, R., ur Rahman, A.: Syn flood dos detection system using time dependent finite automata. International Journal of Computer Science and Network Security **23**(6), 127–132 (2023). https://doi.org/10.22937/IJCSNS.2023.23.6.18
3. Alhazov, A., Freund, R., Ivanov, S., Oswald, M., Verlan, S.: Extended spiking neural P systems with white hole rules. In: Proc. 13th Brainstorming Week on Membrane Computing. pp. 46–62. Sevilla, Spain (2015)
4. Arkko, J., Cotton, M., Vaudreuil, L.: IPv4 Address Blocks Reserved for Documentation. RFC 5737, Internet Engineering Task Force (Jan 2010). https://doi.org/10.17487/RFC5737
5. Bhosale, K.S., Nenova, M., Iliev, G.: A study of cyber attacks: In the healthcare sector. In: 2021 Sixth Junior Conference on Lighting (Lighting). pp. 1–6. Gabrovo, Bulgaria (2021). https://doi.org/10.1109/Lighting49406.2021.9598947
6. Cabarle, F.G.C., Adorna, H.N., Pérez-Jiménez, M.J.: Notes on spiking neural p systems and finite automata. Natural Computing **15** (12 2016). https://doi.org/10.1007/s11047-016-9563-4
7. Cavaliere, M., Ibarra, O.H., Păun, G., Egecioglu, O., Ionescu, M., Woodworth, S.: Asynchronous spiking neural P systems. Theoretical Computer Science **410**(24–25), 2352–2364 (2009). https://doi.org/10.1016/j.tcs.2009.02.031
8. Eddy, W.: Tcp syn flooding attacks and common mitigations. RFC 4987, Internet Engineering Task Force (2007). https://doi.org/10.17487/RFC4987
9. Eddy, W.: Transmission Control Protocol (TCP). RFC 9293, Internet Engineering Task Force (Aug 2022). https://doi.org/10.17487/RFC9293
10. Ferguson, P., Senie, D.: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, Internet Engineering Task Force (May 2000). https://doi.org/10.17487/RFC2827

11. Gulapa, M., Luzada, J.S., Cabarle, F.G.C., Adorna, H.N., Buño, K., Ko, D.: Websnapse reloaded: The next-generation spiking neural p system visual simulator using client-server architecture. In: Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2023). pp. 434–461. Atlantis Press (2024). https://doi.org/10.2991/978-94-6463-388-7_26
12. Han, Z., Wang, X., Wang, F., Wang, Y.: Collaborative detection of DDoS attacks based on Chord protocol. In: 2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012). vol. Supplement, pp. 1–4 (2012). https://doi.org/10.1109/MASS.2012.6708531
13. Haris, S.H.C., Ahmad, R.B., Ghani, M.A.H.A., Waleed, G.M.: Tcp syn flood detection based on payload analysis. In: Proceedings of 2010 IEEE Student Conference on Research and Development (SCOReD 2010). pp. 149–153. IEEE, Putrajaya, Malaysia (Dec 2010). https://doi.org/10.1109/SCORED.2010.5703991
14. Idowu, R.K., Chandren, R.M., Othman, Z.A.: Denial of service attack detection using trapezoidal fuzzy reasoning spiking neural p system. Journal of Theoretical and Applied Information Technology **75**(3), 397–406 (2015)
15. Idowu, R.K., Muniyandi, R.C., Othman, Z.A.: The prospects of using spiking neural p system for intrusion detection. International Journal of Information & Network Security **2**(6), 492–498 (2013). https://doi.org/10.11591/ijins.v2i6.5894
16. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural p systems. Fundamenta informaticae **71**(2-3), 279–308 (2006). https://doi.org/10.3233/fun-2006-712-308
17. Ko, D., Cabarle, F.G.C., De La Cruz, R.T.A.: Websnapse tutorial: A hands-on approach for web and visual simulations of spiking neural p systems. In: Institutional Repository of the University of Seville (2023), https://idus.us.es/items/39409205-d228-41a7-943b-fc5de4bbf328
18. Leporati, A., Mauri, G., Zandron, C.: Spiking neural p systems: main ideas and results. Natural Computing **21**(4), 629–649 (2022). https://doi.org/10.1007/s11047-022-09917-y
19. Luckner, M.: Conversion of decision tree into deterministic finite automaton for high accuracy online syn flood detection. In: 2015 IEEE Symposium Series on Computational Intelligence. pp. 75–82. IEEE (2015). https://doi.org/10.1109/SSCI.2015.21
20. Mukku, B.: A comprehensive study on distributed denial of service attack detection using deep learning technique. In: 2024 First International Conference on Software, Systems and Information Technology (SSITCON). pp. 1–4. Tumkur, India (2024). https://doi.org/10.1109/SSITCON62437.2024.10797085
21. Orellana-Martín, D., Cabarle, F.G.C., Paul, P., Zeng, X., Freund, R.: Wireless spiking neural p systems: D. orellana-martín et al. Journal of Membrane Computing **7**(3), 252–265 (2025)
22. Pleșa, M.I., Gheorghe, M., Ipate, F., Zhang, G.: Applications of spiking neural p systems in cybersecurity. Journal of Membrane Computing **6**(4), 310–317 (2024). https://doi.org/10.1007/s41965-024-00166-9
23. Păun, G.: A quick introduction to membrane computing. The Journal of Logic and Algebraic Programming **79**(6), 291–294 (2010). https://doi.org/10.1016/j.jlap.2010.04.002
24. Scapy Community: Scapy documentation (12 2024), https://scapy.readthedocs.io/en/latest/
25. Scarfone, K., Mell, P.: Guide to intrusion detection and prevention systems (idps). NIST Special Publication 800-94, National Institute of Standards and Technology (2007). https://doi.org/10.6028/NIST.SP.800-94

26. Song, T., Pan, L., Wu, T., Zheng, P., Wong, M.L.D., Rodríguez-Patón, A.: Spiking neural P systems with learning functions. IEEE Transactions on NanoBioscience **18**(2), 176–190 (Apr 2019). https://doi.org/10.1109/TNB.2019.2896981
27. Vasilomanolakis, E., Karuppayah, S., Mühlhäuser, M., Fischer, M.: Taxonomy and survey of collaborative intrusion detection. ACM Computing Surveys **47**(4), 55:1–55:33 (2015). https://doi.org/10.1145/2716260
28. Wang, A., Chang, W., Chen, S., Mohaisen, A.: Delving into internet ddos attacks by botnets: Characterization and analysis. IEEE/ACM Transactions on Networking **26**(6), 2843–2855 (2018). https://doi.org/10.1109/TNET.2018.2874896
29. Wang, J., Hoogeboom, H.J., Pan, L., Păun, G., Pérez-Jiménez, M.J.: Spiking neural p systems with weights. Neural Computation **22**(10), 2615–2646 (2010). https://doi.org/10.1162/NECO_a_00022
30. Wu, T., Pan, L., Yu, Q., Tan, K.C.: Numerical spiking neural P systems. IEEE Transactions on Neural Networks and Learning Systems **32**(6), 2443–2457 (Jun 2021). https://doi.org/10.1109/TNNLS.2020.3005538
31. Zhang, G., Verlan, S., Wu, T., Cabarle, F.G.C., Xue, J., Orellana-Martín, D., Dong, J., Valencia-Cabrera, L., Pérez-Jiménez, M.J.: Software simulations of spiking neural p systems. In: Spiking Neural P Systems: Theory, Applications and Implementations. pp. 317–376. Springer Nature Singapore, Singapore (2024). https://doi.org/10.1007/978-981-97-9282-5_11

# Numerical Networks of Cells with Forward Activation and Blocking of Rules

Artiom Alhazov[1][0000−0002−6184−3971], Rudolf Freund[2][0000−0003−1255−1953], Sergiu Ivanov[3][0000−0002−6286−0917], and Sergey Verlan[4][0000−0002−6286−0917]

[1] Moldova State University,
Vladimir Andrunachievici Institute of Mathematics and Computer Science
Academiei 5, Chișinău, MD-2028, Moldova
artiom@math.md
[2] Faculty of Informatics, TU Wien, Favoritenstraße 9–11, 1040 Wien, Austria
rudi@emcc.at
[3] IBISC, Université Paris-Saclay, 23, boulevard de France, 91020
Èvry-Courcouronnes, France
sergiu.ivanov@univ-evry.fr
[4] LACL, Univ. Paris Est Creteil, 61, av. Général de Gaulle, 94010 Creteil, France
verlan@u-pec.fr

**Abstract.** Numerical networks of cells compute by updating variable values using rules that may be restricted by applicability conditions called guards. Depending on the application mode, the update of a variable either uses the function associated with one applicable rule or sums the values computed by all applicable rules. In this paper, we introduce activation and blocking of rules as an additional control mechanism: a rule may be applied in a computation step only if it is both applicable and activated.

We extend the different application modes to this model and provide illustrative examples, e.g., we show how vector addition systems and register machines can be simulated. Moreover, we prove that the control imposed by the activation and blocking of rules can be represented using additional variables and extended guards. Activation of rules also makes it possible to simulate parallel evolution by sequential steps, provided that the values of unaffected variables are preserved until they are rewritten.

## 1 Introduction

Numerical P systems form a distinctive family within membrane computing. Instead of manipulating multisets of objects, they work with numerical variables whose values evolve in discrete time steps. An important difference from the usual multiset-based interpretation of rules is that these values are shared: reading a variable does not consume it, and the same value may be used simultaneously by several applicable rules. Moreover, several rules may contribute to the value assigned to a variable in the next configuration. These features make numerical

P systems particularly natural for the description of numerical processes and control mechanisms [10,16].

Several extensions of the original model have been proposed. Enzymatic numerical P systems introduced conditions controlling the applicability of rules [9]. This apparently small addition considerably increased the range of behaviors that could be described and supported a number of applications in robotics and control. Examples include membrane controllers for mobile robots [3], trajectory-tracking controllers [15], drone navigation [4], and controllers combining several robotic behaviors [14]. Questions concerning practical implementation have also been investigated, in particular in connection with hardware FPGA-based architectures [11,16].

The notion of a numerical network of cells was introduced in order to place these variants in a more general setting [2]. A numerical network of cells consists of cells containing numerical variables and of rules that compute new values from the current configuration. Guards may restrict the applicability of rules, while producer functions determine their numerical contributions. The framework also makes explicit the distinction between input variables, whose values are provided by the environment, and output variables, whose values may be observed externally. This approach follows the general idea of the formal framework for P systems [6]: rather than defining each new variant from scratch, one identifies a small collection of common ingredients and then studies how changes in these ingredients affect the resulting model. Related frameworks have been developed for P systems with dynamically changing structures [5] and spiking neural P systems [12,13]. Numerical networks of cells play a similar role for models whose configurations are described by numerical values.

In the present paper, we consider an additional way of controlling the use of rules. The main idea is to associate rules with forward activation and blocking information. A rule whose guard is satisfied need not be immediately available: its use may depend on an activation produced in an earlier computation step, or it may be prevented by having applied a rule which now blocks its application. In this way, applicability and availability become two separate notions. Guards describe conditions on the current numerical configuration, whereas activation and blocking describe dependencies between rule applications.

Activation and blocking mechanisms have already been considered in other classes of P systems [1]. Their use in numerical networks of cells is nevertheless of particular interest. Numerical rules often describe different stages of a larger computation, and the order in which these stages become available may be as important as the numerical expressions themselves. Part of this control can be encoded directly into guards, but such encodings usually require auxiliary information and may hide the intended flow of the computation. Explicit activation and blocking relations provide a more direct description of these dependencies.

We call the resulting systems numerical networks of cells with activation and blocking of rules. Besides the components of an ordinary numerical network of cells, such a system specifies which rule applications activate or block other rules and after how many computation steps this influence takes effect. Initial

activations and blockings determine the rules available at the beginning of a computation. We give priority to blocking whenever activation and blocking affect the same rule at the same time. The guards of the rules are retained, so the new mechanism does not replace the usual applicability conditions; rather, it adds a second level of control over the rules that may participate in the current computation step.

The definition is developed for the different rule application modes of numerical networks of cells. We consider all-parallel and sequential evolution, their preserving variants, and strictly sequential modes in which at most one rule is applied in the entire network during a computation step. The activation and blocking mechanism is particularly useful in the strictly sequential setting, where it can be used to schedule individual rule applications and to organize them into longer computational phases.

Several examples illustrate how this control mechanism of activation and blocking of rules can organize numerical computations and enforce dependencies between successive rule applications. For example, we consider vector addition systems, showing how the parallel updates of the vector components can be represented by controlled sequences of rule applications in a preserving strictly sequential setting. Moreover, we also show how register machines can be simulated in such a setting.

We establish several connections between the proposed model and ordinary numerical networks of cells. In particular, we show that activation and blocking can be encoded within the original framework under suitable assumptions, and that this control mechanism can also be used to coordinate simulations between different rule application modes. These results indicate that activation and blocking mainly provide an explicit and convenient control layer rather than a fundamentally different computational mechanism.

The rest of the paper is organized as follows. Section 2 recalls numerical networks of cells and introduces the rule application modes used throughout the paper. Section 3 defines numerical networks of cells with forward activation and blocking of rules. Section 4 presents illustrative examples, in particular the simulation of vector addition systems and register machines. Section 5 gives the simulation and encoding results. Finally, Section 6 concludes the paper.

## 2 Preliminaries

The sets of natural, integer, rational, and real numbers are denoted by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$, respectively. We observe that we assume $\mathbb{N}$ to contain 0; $\mathbb{N} \setminus \{0\}$, the set of positive integers, is denoted by $\mathbb{N}_+$. For two sets $A$ and $B$, we use the notation $A \subset_{fin} B$ to indicate that $A$ is a finite subset of $B$.

Let $(G, +, \mathbf{0})$ be a commutative monoid with the operation $+$ and $\mathbf{0}$ denoting the neutral element of $G$ with respect to $+$. For any $a \in G$ and any $k \in \mathbb{N}$, $ka$ stands for $\sum_{i=0}^{k} a$; observe that $0a = \sum_{i=0}^{0} a = \mathbf{0}$.

For two vectors $u = (u_1, \ldots, u_m) \in G^m$ and $v = (v_1, \ldots, v_k) \in G^k$, we will write $u \oplus v$ for the vector $(u_1, \ldots, u_m, v_1, \ldots, v_k) \in G^{m+k}$.

Now we recall the basic definitions for numerical networks of cells [2].

Let $V$ be a finite alphabet of symbols, called *variables*. A function $F_v : \mathbb{N} \to G$ is associated with each variable $v \in V$; this function can be seen as a discrete-time series $(F_v(t))_{t\in\mathbb{N}}$ giving the value of the variable $v$ at time $t \in \mathbb{N}$. If it is clear from the context we will identify $v$ with $F_v$.

A *cell* groups several variables, i.e., $c = (x_1, \ldots, x_m)$, $m \geq 1$. The *contents* of the cell $c$ at time $t \in \mathbb{N}$ is the vector $c(t) = (x_1(t), \ldots, x_m(t))$. For technical reasons, the cell descriptions are given by the *location* mapping $loc : V \to L$, where $L$ is a set of cell labels and $V$ is the set of variables. We observe that each variable belongs to a single cell. Hence, for each cell $c$ the inverse of $loc$, i.e., $loc^{-1}(c)$, contains the variables associated to $c$.

A *configuration* is a tuple $C(t) = (c_1(t), \ldots, c_n(t))$, $t \geq 0$, where each $c_i$, $i \geq 1$, is a cell, giving the value of all variables at time step $t$. We observe that for a fixed ordering of $V$, the configuration $C(t)$ can be seen as a vector in $G^{|V|}$.

A rule (having the label $r$) is either written as

$$r : g(x_1, \ldots, x_s) \mid f(y_1, \ldots, y_m) \to x$$

or in an alternative notation as

$$r : g(x_1, \ldots, x_s) \mid x := f(y_1, \ldots, y_m)$$

where $g$, called *guard*, is a Boolean mapping $g : G^s \to B$, $f : G^m \to G$ is called the *producer*, and $x \in V$ is the *consumer* variable. If the guard $g$ is always true, $g$ can be omitted. We say that rule $r$ is *applicable* at time $t \geq 0$, if and only if $g(x_1(t), \ldots, x_s(t))$ is true.

By $guard\_vars(r)$ and $prod\_vars(r)$ we denote the set of variables that appear in the guard and the producer of rule $r$, respectively.

Let $R$ be a set of rules. By $R_x$ we denote the set of all rules in $R$ that have the consumer equal to $x$. Moreover, by $Appl(R, C(t))$ we denote the set of all rules from $R$ applicable to the configuration at time $t$, $C(t)$.

We observe that the rules above induce a hypergraph relation among the cells, with hyperedges corresponding to cells involved in a rule: for example, a rule $r : g(x_1, \ldots, x_s) \mid f(y_1, \ldots, y_m) \to x$ corresponds to the hyperedge $(loc(x_1), \ldots, loc(x_s), loc(y_1), \ldots, loc(y_m), loc(x))$.

In the area of P systems usually graphs and trees are considered. Another important structural property is the collocation of all variables from $f$ in the same cell. In this case the corresponding rule is called *local*. It is clear that a local rule can be associated to the cell where all variables of $f$ are from.

We observe that the structure of the system is usually considered as a commodity for applications, as the semantics does not depend on it. In the following, we will not reason about the structure of the system, but only about the variables and their values.

For the application of the rules in $R$ to a given configuration $C(t)$ at time $t$, we can define several semantics.

In the different variants of the *all-parallel* rule application mode the next value of each variable is defined as follows:

$$x(t+1) = \bar{x}(t) + \sum_{r \in Appl(R_x, C(t))} f_r(y_{r,1}(t), \ldots, y_{r,m_r}(t)), \tag{1}$$

where

$$Appl(R_x, C(t)) = \{r_i : g_i(x_{i,1}, \ldots, x_{i,s}) \mid f_i(y_{i,1}, \ldots, y_{i,m}) \to x \mid 1 \leq i \leq k\}.$$

In this mode every applicable rule contributes to the new value of the variable. It is important to note that the current values of variables are not exclusive — they are shared.

We observe that in fact the variable first is set to $\bar{x}(t)$, which function can be defined in different ways:

*allP* In the *zero reset all-parallel rule application mode* (abbreviated *allP*), $\bar{x}(t)$ is always set to $\mathbf{0}$ before the values of all applicable functions are added.

But this means that in the case that no rule in $R_x$ is applicable to $C(t)$, the current value of $x$ gets lost. This mode corresponds to the default semantics of numerical networks of cells, and it is inspired by hardware circuits implementations. However, in some applications it is important to preserve the current value of a variable if no rule is applicable to it. This can be defined in two ways:

*pcallP* In the *preserving contribution all-parallel* rule application mode (abbreviated *pcallP*) the function $\bar{x}(t)$ is defined as follows:

$$\bar{x}(t) = \begin{cases} \mathbf{0} & \text{if } \exists r \in Appl(R, C(t)), \text{with } x \in prod\_vars(r) \\ x(t) & \text{otherwise.} \end{cases} \tag{2}$$

In the *pcallP* mode, the variable is reset to zero before adding the contributions of applicable rules only if it is used in a producer function of an applicable rule. Otherwise, its current value is preserved. This mode corresponds to the default semantics of numerical P systems.

*pallP* In the *preserving all-parallel* rule application mode (abbreviated *pallP*) the function $\bar{x}(t)$ is defined as follows:

$$\bar{x}(t) = \begin{cases} \mathbf{0} & \text{if } Appl(R_x, C(t)) \neq \emptyset, \\ x(t) & \text{if } Appl(R_x, C(t)) = \emptyset. \end{cases} \tag{3}$$

Hence, in the *pallP* mode, if no rule is applicable to a variable, its current value is preserved, otherwise it is reset to zero before adding the contributions of applicable rules.

In the variants of the *sequential* rule application mode, the next value of each variable is defined as follows:

$$x(t+1) = \bar{x}(t) + f(y_1(t), \ldots, y_m(t)) \tag{4}$$

for some $r : g(x_1, \ldots, x_s) \mid f(y_1, \ldots, y_m) \to x \in Appl(R_x, C(t))$.

If there are several applicable rules $r \in R_x$ at time $t$, then one of them is nondeterministically chosen for the evolution. On the other hand, different variables are updated by different applicable rules, but by at most one.

In the case that no rule in $R_x$ is applicable to $C(t)$, the next value of $x$ is set to $\bar{x}(t)$. Again we distinguish between several variants:

*sequ* In the *sequential* rule application mode (abbreviated *sequ*) the function $\bar{x}(t)$ is always set to $\mathbf{0}$.

Again we can define a preserving semantics to avoid losing the current contents of the variable, which will become important when considering activation and blocking of the rules, as we will see in the next sections.

*psequ* In the *preserving sequential* rule application mode (abbreviated *psequ*), as in Equation 3 we take

$$\bar{x}(t) = \begin{cases} \mathbf{0} & \text{if } Appl(R_x, C(t)) \neq \emptyset, \\ x(t) & \text{if } Appl(R_x, C(t)) = \emptyset. \end{cases} \tag{5}$$

The sequential rule application modes *sequ* and *psequ* in fact are only sequential on the level of each variable, but still parallel at the level of all variables in the whole network.

Yet we can also consider sequential modes only using one rule in the overall system. These rule application modes then are called *strictly sequential* and *preserving strictly sequential*, respectively, abbreviated by *ssequ* and *pssequ*. They have not yet been investigated for NNCs, but they will become of more importance together with the control mechanism of activation and blocking of rules.

*Remark 1.* In the all-parallel modes, it is possible to simulate each preserving or zero reset mode by using the other ones. For example, in the zero reset mode it is possible to simulate the preserving mode by adding a rule $r : g_{\neg x}(z_1, \ldots, z_k) \mid x \to x$ with $g_{\neg x}$ being the guard checking for the non-applicability of all other rules with consumer equal to $x$, i.e.,

$$g_{\neg x}(z_1, \ldots, z_k) = \neg \left( \bigvee_{r \in Appl(R_x, C(t))} g_r(z_{r,1}, \ldots, z_{r,s_r}) \right), \tag{6}$$

where $r : g_r(z_{r,1}, \ldots, z_{r,s_r}) \mid f_r(y_{r,1}, \ldots, y_{r,m_r}) \to x$ and the union of all variables from the guards from the above disjunction is $\{z_1, \ldots, z_k\}$.

The simulation works as follows: if there is at least one applicable rule with consumer $x$, then the guard $g_{\neg x}$ is false and the rule $r$ is not applicable. Otherwise, the guard $g_{\neg x}$ is true and the rule $r$ is applicable, keeping the value of $x$ unchanged.

For the preserving contribution all-parallel mode, the simulation is similar, but the guard $g_{\neg x}$ has to check for the non-applicability of all other rules involving $x$ in the producer function:

$$g_{\neg x}(z_1, \ldots, z_k) = \neg \left( \bigvee_{r \in Appl(R, C(t))} g_r(z_{r,1}, \ldots, z_{r,s_r}) \right), \tag{7}$$

where $r : g_r(z_{r,1}, \ldots, z_{r,s_r}) \mid f_r(y_{r,1}, \ldots, y_{r,m_r}) \to y$, with $x$ equal to some $y_{r,i}$. As before, the union of all variables from the guards in the above disjunction is $\{z_1, \ldots, z_k\}$.

Conversely, if we have a preserving mode, we can simulate the zero reset mode by adding a rule $r_{0,x} : x := \mathbf{0}$. This rule is always applicable, and because of its form it will reset the value of $x$ to zero if no other rule is applicable for $x$.

### 2.1 Numerical Networks of Cells

Based on the prerequisites explained above, we now are able to define the model of a numerical network of cells as already defined in [2]:

**Definition 1.** *A* numerical network of cells *(abbreviated by NNC) is a tuple*

$$\Pi = (G, V, L, loc, V(0), Input, Output, R, \delta) \text{ where}$$

- $(G, +, \mathbf{0})$ *is a commutative monoid,*
- $V$ *is a finite alphabet of variables,*
- $L$ *is the set of cell labels,*
- $loc : V \to L$ *is the location function that associates every variable to one specific cell,*
- $V(0)$ *is a vector in* $G^{|V|}$ *that defines the values of all variables at time* $t = 0$*,*
- $Input \subseteq V$ *is the set of ("read-only") input variables, updated by some external entity at each time* $t$*,*
- $Output \subseteq V$ *is the set of ("write-only") output variables that can be read by an external entity at each time* $t$*,*
- $R$ *is a finite set of rules as defined above, and*
- $\delta$ *is one of the rule application modes as defined above.*

The computation of a numerical network of cells $\Pi$ is performed by updating variables, except those defined in the $Input$ set, by using the given semantics $\delta$ as presented above. The variables from the $Input$ set are supposed to be updated in each step by an external entity. An external entity can also read the values of output variables (from the set $Output$) in each time step. These values, $Output(t)$, $t \geq 0$, form the result of the computation of $\Pi$ on the sequence $Input(t)$, $t \geq 0$.

It is obvious that the computation described above in principle is infinite. However, it is also possible to consider finite computations, e.g., if some variable gets a particular value or if some fixed time is elapsed.

In the following, we mostly will consider networks with only one cell, i.e., all variables are located in this single cell. Such a numerical network of cells with only one cell we call a *simple numerical network of cells* and abbreviate it by SNNC. In the description of an SNNC we then omit $L$ and $loc$.

We start with a simple example showing how even the (strictly) sequential modes can be used in a meaningful way.

*Example 1.* We construct an SNNC

$$\Pi = (G = (\mathbb{N}, +, 0), V = \{x, y\}, V(0) = (0, 0), Input = \{x\}, Output = \{y\}, R, \delta)$$

where $R = \{1 : 2x \to y,\ 2 : 3x \to y\}$.

Both rules are always applicable. With $\delta \in \{sequ, psequ, ssequ, pssequ\}$, only one of them is applied in each step; hence, for $y(t+1)$ we either get $2x(t)$ or $3x(t)$, whereas for $\delta \in \{allP, pallP\}$ we only get the single result $5x(t)$. This simple example already shows that with the (strictly) sequential modes also non-determinism comes in.

The condition that output variables cannot be used as inputs for functions updating other variables can be alleviated to make systems less complicated:

*Example 2.* The famous Collatz function $f_C$ is defined for every positive integer $n$ as follows:

- $f_C(n) = n/2$ if $n$ is even and
- $f_C(n) = 3n + 1$ if $n$ is odd.

We now construct an SNNC

$$\Pi = (G = (\mathbb{N}, +, 0), V = \{x, y\}, V(0) = (0, 0), Input = \{x\}, Output = \{y\}, R, \delta)$$

where $R = \{1 : x \to y,\ 2 : g(y) \mid y/2 \to y,\ 3 : \neg g(y) \mid 3y + 1 \to y\}$ and $\delta \in \{pallP, allP\}$. The guard $g(y)$ is true if and only if $y$ is an even number. This corresponds to checking $n = 2 * (n\ DIV\ 2)$, where $DIV$ denotes the integer division.

The input stream for $\Pi$ is assumed to provide $n$ for the first derivation step and to provide only 0 afterwards. Then the external observer of $\Pi$ gets the sequence of integers which are obtained by applying $f_C$ in each step starting with the input $n$, but only after the first step, when the input $n$ has been copied from the input variable $x$ to the output variable $y$ by using rule 1. As the initial value 0 for $y$ is an even number, rule 2 does not add anything to the value of $y$ in the first step. In the succeeding steps, the rules 2 and 3 yield the correct updating of the value of the variable $y$ according to the definition of the Collatz

function. As we have assumed that after having provided the initial value $n$ the input is 0, rule 1 does not contribute to the value of $y$ any more.

We observe that we have used the variable $y$ as input for the functions in rules 2 and 3 and as output variable. If we had strictly obeyed to the original definition, we would have needed a third variable $z$ to be used as output only as well as the rule $4 : y \to z$ for copying the value of the variable $y$ to the output variable $z$.

## 3 Numerical Networks of Cells with Activation and Blocking of Rules

We now equip the numerical network of cells with the control mechanism of activation and blocking of rules. This variant of controlling the application of rules first was introduced in [1].

**Definition 2.** *A* numerical network of cells with activation and blocking of rules *(abbreviated* NNCAB*) is a tuple*

$$\Pi_{AB} = (\Pi, L_R, f_R, \mathcal{A}, \mathcal{B}, \mathcal{A}_0, \mathcal{B}_0),$$

*where*

$$\Pi = (G, V, L, loc, V(0), Input, Output, R, \delta)$$

*is a numerical network of cells, $L_R$ is a finite set of labels with each label having assigned one rule from $R$ by the function $f_R$,*

$$\mathcal{A}, \mathcal{B} \subset_{fin} L_R \times L_R \times \mathbb{N}_+$$

*are the relations of activations and blockings of rules, respectively, and*

$$\mathcal{A}_0, \mathcal{B}_0 \subset_{fin} L_R \times \mathbb{N}$$

*are the initial activations and blockings, respectively; we require $\mathcal{A}_0 \cap \mathcal{B}_0 = \emptyset$.*

For rule labels $l_p, l_q \in L_R$ and $t \in \mathbb{N}_+$, the triple $(l_p, l_q, t) \in \mathcal{A}$ means that the application of the rule $p$ labeled by $l_p$ activates the rule $q$ labeled by $l_q$ in the future step that is $t$ steps ahead of the current one, while $(l_p, l_q, t) \in \mathcal{B}$ means that (the application of) $p$ blocks $q$ for step $t$ ahead.

For a rule $p$ labeled by $l_p \in R$ and $t \in \mathbb{N}$, the pair $(l_p, t) \in \mathcal{A}_0$ means that this rule $p$ is *activated* for the derivation step that will be carried out $t$ steps ahead in the future, while $(l_p, t) \in \mathcal{B}_0$ means that $p$ is *blocked* for step $t$ ahead. We mention that $t = 0$ indicates the activations for the first step to be carried out.

In the following, we will not distinguish between saying *the rule labeled by $r$* and saying *the rule $r$* to avoid overfluous wording, as it should always be clear from the context what we are talking about.

A derivation in $\Pi_{AB}$ starts with the initial configuration and using the activated and blocked rules in $\mathcal{A}_0$ and $\mathcal{B}_0$, respectively, in the derivation mode $\delta$. At any time $t$ of a derivation, we have a set of activated rules $Q(t)$ and blocked rules $\bar{Q}(t)$ with $Q(t) \cap \bar{Q}(t) = \emptyset$; thus, we also have required $\mathcal{A}_0 \cap \mathcal{B}_0 = \emptyset$. Moreover, we assume blocking to always have priority over activation. As $\mathcal{A}_0$ and $\mathcal{B}_0$, also $Q(t)$ and $\bar{Q}(t)$ are finite subsets of $L_R \times \mathbb{N}$. The set of rules available for the first derivation step in the given derivation mode consists of all rules $p$ labeled by $r_p$ such that $(r_p, 0) \in \mathcal{A}_0$.

As the applicability of the rules in $R$ also depends on the guards, we will call the rules $p$ labeled by $r_p$ such that $(r_p, 0) \in Q(t)$ *licensed* for a configuration $C(t)$ in the NNCAB $\Pi$; the set of all licensed rules for a configuration $C(t)$ is denoted by $Lic(R, C(t))$. If moreover $p \in Appl(R, C(t))$, i.e., if $p \in Appl(R, C(t)) \cap Lic(R, C(t))$, then $p$ is called *active* for $C(t)$, and we write $r_p \in Act(C(t))$.

We emphasize that for all the conditions concerning the applications of rules as described in the preceding sections instead of the whole set of rules $R$ we now have to take the set of licensed rules $Lic(R, C(t))$.

For example, Equation 6 now reads as

$$g_{\neg x}(z_1, \ldots, z_k) = \neg \left( \bigvee_{p \in Appl(Lic(R,C(t))_x, C(t))} g_p(z_{p,1}, \ldots, z_{p,s_p}) \right). \qquad (8)$$

Moreover, we observe that $p \in Appl(Lic(R, C(t))_x, C(t))$ can be replaced by $r_p \in Act(C(t))_x$. Yet such a condition cannot be checked with just looking at the current configuration only given by the underlying NNC. Hence, the additional information has to be encoded in another way, for example, by using an additional variable which keeps track of the activations and blockings still relevant for the current and future derivation steps, which idea, for example, will be used in the proof of Theorem 2..

In the all-parallel modes, all active rules are applied to $C$. Hence, the sets $Q(t)$ and $\bar{Q}(t)$ are updated as follows:

$$\begin{aligned} \bar{Q}(t+1) &= \{(r, i-1) \mid (r, i) \in \bar{Q}(t),\ i \geq 1\} \\ &\quad \cup \{(r, i-1) \mid (q, r, i) \in \mathcal{B},\ q \in Act(C(t))\}, \\ Q(t+1) &= (\{(r, i-1) \mid (r, i) \in Q(t),\ i \geq 1\} \\ &\quad \cup \{(r, i-1) \mid (q, r, i) \in \mathcal{A},\ q \in Act(C(t))\}) \\ &\quad \setminus \{(r, j) \mid (r, j) \in \bar{Q}(t+1)\} \end{aligned}$$

In the sequential modes, only one of the active rules for each variable is applied to $C$. Let the label of the active rule chosen for the variable $x$ be $q_{s,x}$, i.e., for each $x \in V$, $q_{s,x}$ must be in $Act(C(t))$. Then the sets $Q(t)$ and $\bar{Q}(t)$ are updated as follows:

$$\begin{aligned}\bar{Q}(t+1) &= \big\{(r,i-1) \mid (r,i) \in \bar{Q}(t),\ i \geq 1\big\} \\ &\quad \cup \big\{(r,i-1) \mid (q_{s,x},r,i) \in \mathcal{B}, x \in V\big\}, \\ Q(t+1) &= (\{(r,i-1) \mid (r,i) \in Q(t),\ i \geq 1\} \\ &\quad \cup\ \{(r,i-1) \mid (q_{s,x},r,i) \in \mathcal{A}, x \in V\}) \\ &\quad \setminus\ \big\{(r,j) \mid (r,j) \in \bar{Q}(t+1)\big\}\end{aligned}$$

In the strictly sequential modes, only one of the active rules is applied to $C$. Let the label of this rule be $q_s$. Then the sets $Q(t)$ and $\bar{Q}(t)$ are updated as follows:

$$\begin{aligned}\bar{Q}(t+1) &= \big\{(r,i-1) \mid (r,i) \in \bar{Q}(t),\ i \geq 1\big\} \\ &\quad \cup \big\{(r,i-1) \mid (q_s,r,i) \in \mathcal{B}\big\}, \\ Q(t+1) &= (\{(r,i-1) \mid (r,i) \in Q(t),\ i \geq 1\} \\ &\quad \cup\ \{(r,i-1) \mid (q_s,r,i) \in \mathcal{A}\}) \\ &\quad \setminus\ \big\{(r,j) \mid (r,j) \in \bar{Q}(t+1)\big\}\end{aligned}$$

We observe that in all cases $Q(t+1)$ and $\bar{Q}(t+1)$ are made non-conflicting by eliminating labels of rules which would be activated and blocked at the same time from the set of activated rules $Q(t+1)$.

In general, we allow the system to make idle steps without applying any rule, provided that still some rules are activated for forthcoming steps; otherwise, we say that the system *halts*.

*Remark 2.* We observe that the concept of *halting* now comes in even if we have an arbitrary stream for the input variables, whereas in an NCC rules may become applicable again depending on the inputs.

*Remark 3.* If the rules in the underlying NNC are already labeled, we can omit $L_R$ and $f_R$ in the definition of the NNCAB.

*Remark 4.* If the underlying NNC of an NNCAB is simple, the NNCAB is called simple, too, abbreviated SNNCAB. If no blockings of rules are used in an NCCAB/SNCCAB, we only write NCCA/SNCCA and speak of a *numerical network of cells with activation of rules*. We then also omit $\mathcal{B}$ and $\mathcal{B}_0$ in the description of the system.

## 4 Expressive Power

In this section we illustrate the expressive power of (simple) numerical networks of cells with activation and blocking of rules by specific examples, in particular showing how vector addition systems and register machines can be simulated.

Our first example shows how activations can control the output:

*Example 3.* We again consider the SNNC from Example 1

$$\Pi = (G = (\mathbb{N}, +, 0), V = \{x, y\}, V(0) = (0, 0), Input = \{x\}, Output = \{y\}, R, \delta)$$

where $R = \{1 : 2x \to y,\ 2 : 3x \to y\}$ and embed it into the numerical network of cells with activation (and blocking) of rules

$$\Pi_{A_1} = (\Pi, \mathcal{A} = \{(1, 2, 1), (2, 1, 1)\}, \mathcal{B} = \emptyset, \mathcal{A}_0 = \{(1, 0)\}, \mathcal{B}_0 = \emptyset).$$

As $\mathcal{B} = \mathcal{B}_0 = \emptyset$ , we simply write

$$\Pi_{A_1} = (\Pi, \mathcal{A} = \{(1, 2, 1), (2, 1, 1)\}, \mathcal{A}_0 = \{(1, 0)\}).$$

We now only consider $\delta \in \{ssequ, pssequ\}$. As $\mathcal{A}_0 = \{(1, 0)\}$, in the first step rule 1 is applied. Afterwards, in even steps rule 2 and in odd steps rule 1 is applied. Hence, alternately, for $y(t + 1)$ we get $2x(t)$ and $3x(t)$, respectively. In every step, now exactly one output is possible, hence, the system works in a deterministic way.

We now return to our example simulating the Collatz function:

*Example 4.* A simple numerical network of cells only using activation of rules based on the SNNC described in Example 2 is defined as

$$\Pi_A = (\Pi, \mathcal{A}, \mathcal{A}_0 = \{(1, 0)\})$$

with the SNNC

$$\Pi = (G = (\mathbb{N}, +, 0), V = \{x, y\}, V(0) = (0, 0), Input = \{x\}, Output = \{y\}, R, \delta)$$

where $R = \{1 : x \to y,\ 2 : g(y) \mid y/2 \to y,\ 3 : \neg g(y) \mid 3y + 1 \to y\}$.

As modes $\delta$ we now can take the all-parallel modes $pallP$ and $allP$, but also the sequential modes $sequ$ and $psequ$ as well as the strictly sequential modes $ssequ$ and $pssequ$. Moreover, we take the activations

$$\mathcal{A} = \{(1, 2, 1), (1, 3, 1), (2, 2, 1), (2, 3, 1), (3, 2, 1), (3, 3, 1)\}.$$

$\mathcal{A}_0 = \{(1, 0)\}$ means that we can only start by applying rule 1, which means taking over the initial value from the environment into variable $y$. Then both rule 2 and rule 3 are activated, but according to the guards, exactly one of these two rules can be applied. In each succeeding steps, again both rule 2 and rule 3 are activated, but according to the guards, exactly one of these two rules can be applied.

The advantage of $\Pi_A$ in contrast to $\Pi$ is that now also the (strictly) sequential modes yield the desired output stream. Moreover, as rule 1 is never activated again, the rest of the input stream after having provided the input $n$ is irrelevant.

We again observe that we have used the variable $y$ as input for the functions in rules 2 and 3 and as output variable. If instead we would like to use a third variable $z$ as output variable, the rule $4 : y \to z$ for copying the value of the variable $y$ to the output variable $z$ has to be used. Now only the all-parallel modes and the preserving sequential modes can be used. We have $V(0) = (0, 0, 0)$ and, moreover, the set of activations has to be extended:

– for the modes $allP$ and $pallP$:
$\mathcal{A} = \{(1,2,1),(1,3,1),(1,4,1),(2,2,1),(2,3,1),(2,4,1),$
$(3,2,1),(3,3,1),(3,4,1)\}$;
– for the modes $psequ, pssequ$:
$\mathcal{A} = \{(1,4,1),(4,2,1),(4,3,1),(2,4,1),(3,4,1)\}$.

It is interesting to see that in the preserving (strictly) sequential mode the computations follow a clear path: After the application of any of the rules in $\{1,2,3\}$, rule 4 is applied, whereafter the rules 2 and 3 are activated again. On the other hand, in these modes each update takes two steps, i.e., it takes two steps before the new output appears in the output variable $z$.

*Remark 5.* We observe that the rules of $R$ in the previous example are already labeled, hence, according to Remark 3, we have omitted specifying the set of labels $\{1,2,3\}$ for the rules as well as the bijective function assigning the labels to the rules. We mention that also in the rest of the paper we usually will directly label the rules in $R$ and omit $L_R$ and $f_R$ in the definition of an NNCAB.

### 4.1 Vector Addition Systems

We now show how vector addition systems can be simulated.

**Definition 3.** *Let $G' = (G, +, \mathbf{0})$ be a commutative monoid. We define a* vector addition system*, abbreviated VAS, over $G$ of* degree *$d$ and* size *$k$ as a tuple $(G', v_0, \ldots, v_d)$, where the $v_j$, $0 \leq j \leq d$, are $k$-dimensional vectors over $G$; $v_0$ is the initial vector. In every computation step one of the other vectors is added. The resulting sequence of $k$-vectors over $G$ is considered as the output of the system.*

*Remark 6.* We mention that the standard definition takes the commutative monoid $(\mathbb{Z}, +, \mathbf{0})$, but as an additional condition only resulting vectors with all components being in $\mathbb{N}$ are taken. We will come back to this special variant later.

**Theorem 1.** *For every vector addition system $\Pi_{VAS} = (G', v_0, \ldots, v_d)$ over a commutative monoid $G' = (G, +, \mathbf{0})$ of degree $d$ and size $k$ we can construct an SNNCA simulating $\Pi_{VAS}$ with $k$ variables in the rule application mode pssequ.*

*Proof.* For each component $i$ of the $k$-vectors we take one corresponding variable $x_i$, which also works as an output variable. In every derivation step, exactly one activated rule will be used for just one variable. The task of the activations is to control the sequence of these rules representing the addition of one of the vectors $v_j$, $1 \leq j \leq d$. In the first step, one of these sequences of rules is activated by using one of the functions for the first variable.

For the vector addition system $\Pi_{\mathrm{VAS}} = (G', v_0, \ldots, v_d)$ we define the corresponding SNNCA $\Pi_A$ as follows:

$$\Pi_A = (\Pi, \mathcal{A}, \mathcal{A}_0) \text{ where}$$

$$\Pi = (G', V = \{x_i \mid 1 \leq i \leq k\}, V(0) = v_0, Input = \emptyset, Output = V, R, pssequ)$$

with $R = \bigcup_{1 \leq j \leq d, 1 \leq i \leq k} \{r_{i,j}\}$ and $\mathcal{A}_0 = \{(r_{1,m}, 0) \mid 1 \leq m \leq d\}$.

The rules $r_{i,j}$ are defined by $r_{i,j} : x_i := x_i + v_j(i)$. To coordinate the additions, we define the following activations to be taken into $\mathcal{A}$:

- $(r_{i,j}, r_{i+1,j}, 1),\ 1 \leq j \leq d,\ 1 \leq i \leq k-1$;
- $(r_{k,j}, r_{1,m}, 1),\ 1 \leq j, m \leq d$.

At the end of simulating the addition of one vector, the first rule for the addition of the next vector is initiated. A reasonable result vector in the output variables can be read after each cycle of $k$ steps. □

*Example 5.* In the standard vector addition system over the commutative monoid $(\mathbb{Z}, +, \mathbf{0})$, the additional condition is to allow only resulting vectors with all components being in $\mathbb{N}$. Therefore, when constructing a simulating SNNCA $\Pi_A$ as in the proof of Theorem 1, we have to use suitable guards for the rules $r_{1,m}$, $1 \leq m \leq d$, which have to guarantee the additional restriction that only vectors with all components being in $\mathbb{N}$ are generated. This can be accomplished by using the guards $g_{1,j} = \wedge_{1 \leq i \leq k}(x_i + v_j(i) \geq 0)$, yielding the amended rules $r_{1,j} : g_{1,j} \mid x_1 := x_1 + v_j(1)$, for $1 \leq j \leq d$.

**Definition 4.** *Let $G' = (G, +, \mathbf{0})$ be a commutative monoid. We define a* vector addition system with states*, abbreviated VASS, over $G$ of* degree $d$ *and* size $k$ *as a tuple $(S, C)$ where $S = (G', v_0, \ldots, v_d)$ is a VAS and $C$ is a* control graph *with $C = (Q, \tau, Q_0)$, $Q$ is the set of states, $\tau \subseteq Q \times W \times Q$ is the set of transitions, where $W = \{v_1, \ldots, v_d\}$, and $Q_0$ is the set of initial states. A computation starts with a configuration $(s_0, v_0)$, for some $s_0 \in Q_0$. A computation step leads from a configuration $(s, v)$ to a configuration $(s', v')$ if and only if $v' = v + u$ for some $(s, u, s')$. The resulting sequence of $k$-vectors over $G$ is considered as the output of the system. In the case of a VASS over $(\mathbb{Z}, +, \mathbf{0})$, the additional condition again is that only resulting vectors with all components being in $\mathbb{N}$ are allowed.*

**Theorem 2.** *For every VASS $(S, C)$ with $S = (G', v_0, \ldots, v_d)$ and the control graph $C = (Q, \tau, Q_0)$ over an infinite commutative monoid $G' = (G, +, \mathbf{0})$ of degree $d$ and size $k$ we can construct an SNNCA simulating $(S, C)$ with $k + 1$ variables in the rule application mode pssequ.*

*Proof.* For each component $i$ of the $k$-vectors again we take one corresponding variable $x_i$, which also works as an output variable. Yet we use an additional variable $x_0$ for simulating the control given by $C$. As the number of states and transitions in a VASS in general is not bounded, we had to make the assumption that $G$ is infinite, as different transitions will be encoded by different elements of $G$.

Thus, let $Q = \{q_s \mid 1 \leq s \leq m\}$. Then we take a subset of different elements

$$G_\tau = \{a_{s,j,s'} \mid 1 \leq s, s' \leq m, 1 \leq j \leq d, (s, v_j, s') \in \tau\}$$

of $G$, $\mathbf{0} \notin G_\tau$, with $a_{s,j,s'}$ representing the transition $(s, v_j, s') \in \tau$.

For the VASS $(S, C)$ with $S = (G', v_0, \ldots, v_d)$ and $C = (Q, \tau, Q_0)$, we now define the corresponding SNNCA $\Pi_A$ as follows:

$$\Pi_A = (\Pi, \mathcal{A}, \mathcal{A}_0) \text{ where}$$

$$\Pi = (G', V = \{x_i \mid 0 \leq i \leq k\}, V(0) = (0) \oplus v_0, Input = \emptyset, Output = V, R, pssequ)$$

where

- $R = R_0 \cup R'$,
  $R_0 = \{r_{0,(s,j,s')} \mid 1 \leq s, s' \leq m, 1 \leq j \leq d, (s, v_j, s') \in \tau, s \in Q_0\}$,
  $R' = \bigcup_{1 \leq s,s' \leq m, 1 \leq j \leq d, 1 \leq i \leq k} \{r_{s,j,s',i}\}$;
- $r_{0,(s,j,s')} : x_0 = 0 \mid x_0 := a_{s,j,s'}, \quad 1 \leq s, s' \leq m, \, (s, v_j, s') \in \tau, s \in Q_0$;
- $r_{s,j,s',i} : x_0 = a_{s,j,s'} \mid x_i := x_i + v_j(i), 1 \leq s, s' \leq m, \, (s, v_j, s') \in \tau, 1 \leq i \leq k$.

In every derivation step, exactly one activated rule will be used for just one variable. The task of the activations is to control the sequence of the rules representing the addition of one of the vectors $v(i)$, but now according to the transitions given by the relation $\tau$ of the control graph $C$.

In the first step, one of these sequences of rules has to be started from an initial state in $Q_0$. For being able to simulate a start from any state in $Q_0$, we use the rules $r_{0,s,j,s'}$ and the initial activations

$$\mathcal{A}_0 = \{(r_{0,(s,j,s')}, 0) \mid 1 \leq s, s' \leq m, \; (s, v_j, s') \in \tau, \; s \in Q_0\}.$$

The additions of the components of the vector $v_j$ with $(s, v_j, s') \in \tau$ are simulated by the following activations to be taken into $\mathcal{A}$:

- $(r_{s,j,s',i}, r_{s,j,s',i+1}, 1), \; 1 \leq i \leq k-1$;
- $(r_{s,j,s',k}, r_{s',\ell,s'',1}, 1), \; 1 \leq j, \ell \leq d, \, (s, v_j, s'), (s', v_\ell, s'') \in \tau$.

At the end of simulating the addition of one vector, the first rule for the addition of the next vector according to the transition $(s', v_\ell, s'') \in \tau$ is initiated.

We finally observe that when also taking $x_0$ as an output variable, we are able to keep track of what happens in each derivation step. On the other hand, if we are only interested in the updated vectors as the results, we can exclude $x_0$ and only consider the values of the output variabes $\{x_i \mid 1 \leq i \leq k\}$ in cycles of $k$ steps. □

*Example 6.* For a standard VASS over the commutative monoid $(\mathbb{Z}, +, \mathbf{0})$, the additional condition to allow only resulting vectors with all components being in $\mathbb{N}$ can be implemented in a similar way as in Example 6. Yet we omit the detailed construction here, as it is well known, see [7], that three additional components in the vectors, i.e., three additional variables, are sufficient to simulate a VASS by a VAS over $(\mathbb{Z}, +, \mathbf{0})$. On the other hand, the advantage of a direct construction is that only one additional variable is needed instead of three, yet instead using the additional control mechanism of activation of rules.

### 4.2 Register Machines

Register machines are well-known universal devices for computing on (or generating or accepting) sets of vectors of natural numbers, e.g., see [8].

**Definition 5.** *A* register machine *is a construct* $M = (n, B, l_0, l_h, P)$ *where* $n$ *is the number of registers,* $P$ *is the set of* instructions *bijectively labeled by elements of* $B$*,* $l_0 \in B$ *is the* initial label*, and* $l_h \in B$ *is the* final label*.*

*The instructions of* $M$ *are of the following forms:*

- $p : (ADD(r), q, s)$*;* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq n$*.*
  *Increase the value of register* $r$ *by one, and non-deterministically jump to instruction* $q$ *or* $s$*.*
- $p : (SUB(r), q, s)$*;* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq n$*.*
  *If the value of register* $r$ *is not zero, then decrease the value of register* $r$ *by one (*decrement *case) and jump to instruction* $q$*, otherwise jump to instruction* $s$ *(*zero-test *case).*
- $l_h : HALT$*.*
  *Stop the execution of the register machine.*

*The register affected by the instruction (labeled by)* $p$ *is denoted by* $reg(p)$*, and* $instr(p)$ *denotes the instruction ADD or SUB in* $p$*.*

*A* configuration *of a register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction to be executed.*

For useful results on the computational power of register machines, we refer to [8]; for example, register machines can generate all recursively enumerable sets of vectors of natural numbers with $k$ components using precisely $k+2$ registers, with only two registers ever using *SUB*-instructions.

In the most general case of computing relations, specific registers of the register machine $M$ are declared as *input registers*, others as *output registers*.

A special variant of register machines are *blind register machines*, where registers are allowed to take values from $\mathbb{Z}$ and not only from $\mathbb{N}$, i.e., decrements may lead to values less than zero.

**Definition 6.** *A* blind register machine *is a construct* $M = (n, B, l_0, l_h, P)$ *where* $n$ *is the number of registers,* $P$ *is the set of* instructions *bijectively labeled by elements of* $B$*,* $l_0 \in B$ *is the* initial label*, and* $l_h \in B$ *is the* final label*, as in register machines, but the instructions of* $M$ *now are of the following forms:*

- $p : (ADD(r), q, s)$*;* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq n$*.*
  *Increase the value of register* $r$ *by one, and non-deterministically jump to instruction* $q$ *or* $s$*.*
- $p : (SUB(r), q)$*;* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq n$*.*
  *The value of register* $r$ *is decreased by one, and the machine jumps to instruction* $q$*.*

– $l_h : HALT$.
*Stop the execution of the register machine.*

At the end of a successful derivation, all registers which are not output registers must be zero.

We now prove that register machines can be simulated by SNNCAs over $(\mathbb{N}, +, \mathbf{0})$ in a strictly sequential rule application mode. Instead of the usual subtract operation used in $(\mathbb{Z}, +, \mathbf{0})$, for $(\mathbb{N}, +, \mathbf{0})$ we use the operation $\ominus$, which is defined for all $x, y \in \mathbb{N}$ by

$$x \ominus y = \begin{cases} x - y \text{ if } x \geq y, \\ 0 \text{ otherwise.} \end{cases}$$

**Theorem 3.** *For every register machine $M = (n, B, l_0, l_h, P)$ with $n$ registers we can construct an SNNCA with $n$ variables which simulates the computations of $M$ in real time when working in the preserving strictly sequential rule application mode pssequ.*

*Proof.* Given $M$, the SNNCA $\Pi_A$ is constructed as follows:

$$\Pi_A = (\Pi, \mathcal{A}, \mathcal{A}_0) \text{ where}$$

– $\Pi = ((\mathbb{N}, +, \mathbf{0}), V, V(0), Input = V_{in}, Output = V_{out}, R, pssequ)$;
– $V = \{x_i \mid 1 \leq i \leq n\}$;
– $V(0) = v_0$, where $v_0$ represents the initial contents of the registers; all registers not being input registers, by default, are assumed to be 0 at the beginning;
– $V_{in} = \emptyset$, as we assume the input values being directly given in $v_0$;
– $V_{out}$ contains the variables for the output registers;
– for simulating the instructions of $M$, the following rules are taken into $R$ and the following activations are taken into $\mathcal{A}$:
  1. for $p : (ADD(r), q, s)$, $p, q, s \in B \setminus \{l_h\}$, $1 \leq r \leq n$:
  $r_{p,q} : x_r := x_r + 1$;
  the activations $(r_{p,q}, r_{q,u}, 1)$ and $(r_{p,q}, r_{q,v}, 1)$ for $q : (instr(q)(reg(q)), u, v) \in P$ are taken into $\mathcal{A}$;
  $r_{p,s} : x_r := x_r + 1$;
  the activations $(r_{p,s}, r_{s,u}, 1)$ and $(r_{p,s}, r_{s,v}, 1)$ for $s : (instr(s)(reg(s)), u, v) \in P$ are taken into $\mathcal{A}$;
  2. for $p : (SUB(r), q, s)$, $p, q, s \in B \setminus \{l_h\}$, $1 \leq r \leq n$:
  $r_{p,q} : x_r > 0 \mid x_r := x_r \ominus 1$;
  the activations $(r_{p,q}, r_{q,u}, 1)$ and $(r_{p,q}, r_{q,v}, 1)$ for $q : (instr(q)(reg(q)), u, v) \in P$ are taken into $\mathcal{A}$;
  $r_{p,s} : x_r = 0 \mid x_r := x_r$;
  the activations $(r_{p,s}, r_{s,u}, 1)$ and $(r_{p,s}, r_{s,v}, 1)$ for $s : (instr(s)(reg(s)), u, v) \in P$ are taken into $\mathcal{A}$;

3. $l_h : HALT$: for the Halt-instruction, no rule is needed, as $M$ halts as soon as $l_h$ is reached, and also the simulating SNNCA will halt in this case, as no further rule is activated, because in the cases 1. and 2. above no activations are taken into $\mathcal{A}$ if $q$ or $s$ equal $l_h$;

– to start the simulation correctly, for $l_0 : (instr(l_0)(reg(l_0)), q, s)$ we need the following activations in $\mathcal{A}_0$: $(r_{l_0,q}, 0)$ and $(r_{l_0,s}, 0)$.

The preserving strictly sequential mode *pssequ* allows for a real-time simulation of the computations of $M$ by the SNCCA $\Pi_A$; moreover, $\Pi_A$ halts if and only if $M$ halts. □

The construction of an SNCCA for simulating a blind register machine is much easier, as the *SUB*-instructions now can be carried out over $\mathbb{Z}$ without any restriction. Only at the end the condition that all registers which are not output registers have to be zero needs some extra effort.

**Theorem 4.** *For every blind register machine $M = (n, B, l_0, l_h, P)$ with $n$ registers we can construct an SNNCA with $n$ variables which simulates the computations of $M$ in real time when working in the strictly sequential rule application mode pssequ.*

*Proof.* Given $M$, the SNNCA $\Pi_A$ is constructed as follows:

$$\Pi_A = (\Pi, \mathcal{A}, \mathcal{A}_0) \text{ where}$$

– $\Pi = ((\mathbb{Z}, +, \mathbf{0}), V, V(0), Input = \emptyset, Output = V_{out}, R, pssequ)$;
– $V = \{x_i \mid 1 \leq i \leq n\}$;
– $V(0) = v_0$, where $v_0$ represents the initial contents of the input registers of $M$;
– $Output = V_{out}$ contains the variables for the output registers;
– for simulating the instructions of $M$, the following rules are taken into $R$ and the following activations are taken into $\mathcal{A}$:
  1. for $p : (ADD(r), q, s)$, $p \in B \setminus \{l_h\}$, $q, s \in B$, $1 \leq r \leq n$:
     $r_p : x_r := x_r + 1$;
     the activations $(r_p, r_q, 1)$ and $(r_p, r_s, 1)$ are taken into $\mathcal{A}$;
  2. for $p : (SUB(r), p)$, $p \in B \setminus \{l_h\}$, $q \in B$, $1 \leq r \leq n$:
     $r_p : x_r := x_r - 1$;
     the activation $(r_p, r_q, 1)$ is taken into $\mathcal{A}$;
  3. $l_h : HALT$: for the Halt-instruction, the corresponding rule has to check that the variables in $V \setminus V_{out}$ are zero; only in this case, this rule $r_h$ is *not* applied, i.e., the SNNCA halts, whereas otherwise this rule is applied forever:

     $r_h : \bigvee_{x_i \in V \setminus V_{out}} x_i \neq 0 \mid x_1 := x_1$;
– to start the simulation correctly, we take the activation $(l_0, 0)$ into $\mathcal{A}_0$.

Again the preserving strictly sequential mode *pssequ* allows for a real-time simulation of the computations of $M$ by the SNCCA $\Pi_A$; moreover, $\Pi_A$ halts if and only if $M$ halts and in addition all registers not being output registers are zero. □

The following construction shows how register machines can be simulated in an easier way when using blocking of rules, but at the cost of needing three steps for simulating a *SUB*-instruction.

**Theorem 5.** *For every register machine $M = (n, B, l_0, l_h, P)$ with $n$ registers we can construct an SNNCAB with $n$ variables which simulates the computations of $M$ working in the preserving strictly sequential rule application mode pssequ, without using guards checking a variable for zero.*

*Proof.* Given $M$, the SNNCAB $\Pi_{AB}$ is constructed as follows:

$$\Pi_{AB} = (\Pi, \mathcal{A}, \mathcal{B}, \mathcal{A}_0, \mathcal{B}_0 = \emptyset) \text{ where}$$

- $\Pi = ((\mathbb{N}, +, \mathbf{0}), V, V(0), Input = \emptyset, Output = V_{out}, R, pssequ)$;
- $V = \{x_i \mid 1 \leq i \leq n\}$;
- $V(0) = v_0$, where $v_0$ represents the initial contents of the input registers of $M$;
- $Output = V_{out}$ contains the variables for the output registers;
- for simulating the instructions of $M$, the following rules are taken into $R$ and the following activations and blockings are taken into $\mathcal{A}$ and $\mathcal{B}$, respectively:
  1. for $p : (ADD(r), q, s)$, $p \in B \setminus \{l_h\}$, $q, s \in B$, $1 \leq r \leq n$:
     $r_p : x_r := x_r + 1$;
     the activations $(r_p, r_q, 1)$ and $(r_p, r_s, 1)$ are taken into $\mathcal{A}$;
  2. for $p : (SUB(r), q, s)$, $q \in B \setminus \{l_h\}$, $q, s \in B$, $1 \leq r \leq n$:
     $r_p : x_r := x_r$;
     the activations $(r_p, r'_p, 1)$ and $(r_p, r_s, 2)$ are taken into $\mathcal{A}$;
     $r'_p : x_r > 0 \mid x_r := x_r \ominus 1$;
     the activation $(r_p, r_q, 1)$ is taken into $\mathcal{A}$ and the blocking $(r_p, r_s, 1)$ is taken into $\mathcal{B}$;
- to start the simulation correctly, we take the activation $(l_0, 0)$ into $\mathcal{A}_0$.

The simulation of a *SUB*-instruction now takes three steps: the first step is only used to activate the instruction for the zero-test case, two steps ahead, which is blocked if in the step before the decrement rule $r'_p$ was applicable. □

## 5 General Results

In this section we show several general results for NCCABs. The first result seems to be obvious, but some technical details have to be taken into account for all rule application modes, because in case no rule is applicable at some moment, an additonal rule must guarantee that the system can continue with its computation as soon as new input changes the situation and some rules may become applicable again.

**Theorem 6.** *Every NCC working in the rule application mode $\delta$ can be simulated by an NCCA also working in the rule application mode $\delta$, thereby using one additional variable and just one rule for this variable.*

*Proof.* Given an NCC $\Pi' = (G, V', L, loc, V'(0), Input, Output, R', \delta)$ we construct the corresponding NNCA

$$\Pi_A = (\Pi, L_R, f_R, \mathcal{A}, \mathcal{B} = \emptyset, \mathcal{A}_0, \mathcal{B}_0 = \emptyset),$$

where

$$\Pi = (G, V' \cup \{z\}, L \cup \{l_z\}, loc \cup \{(z, l_z\}, V'(0) \oplus (\mathbf{0}), Input, Output, R, \delta)$$

is an NCC with the additional variable $z$, which has the initial value $\mathbf{0}$ and will always keep the value $\mathbf{0}$.

$L_R$ is a finite set of labels with each label having assigned one rule from $R$ by the function $f_R$. The set of rules $R = R' \cup \{r_z\}$ extends $R'$ by the rule

$$r_z : g_z \mid z \to \mathbf{0} \quad \text{with} \quad g_z = \neg \left( \bigvee_{r \in R'} g_r \right),$$

which means that $r_z$ can only be applied if no other rule is applicable. The application of $r_z$ guarantees that even if no rule from $R'$ was applicable, in the next step all rules are available again, as we define

$$\mathcal{A} = \{(r, s, 1) \mid r, s \in R\}$$

as well as

$$\mathcal{A}_0 = \{(r, 0) \mid r \in R\},$$

which guarantees that all rules are activated at the beginning. □

The next theorem shows that activation and blocking can be simulated by using an additional variable and suitable guards, provided the underlying commutative monoid $G$ is infinite.

**Theorem 7.** *Any NCCAB over an infinite commutative monoid $G$ working in the rule application mode $\delta$ can be simulated by an NCC also working in the same rule application mode $\delta$, thereby using only one additional variable.*

*Proof. (Sketch)* The additional variable can be used to control which rules are licensed for the next steps. The original guards are extended by a logic expression $z = a$, where $a \in G$ represents the current *state* of the system encoding the current sets $Q(t)$ and $\bar{Q}(t)$. This state variable then evolves by functions $g \mid b \to z$, where the guard $g$ includes the logic expression $z = a$ as well as the conjunction of the guards of the applied functions according to the given mode $\delta$; for example, see the construction given in Example 5. □

A complete proof for the preceding and the succeeding theorem is left for an extended journal version, or the interested reader.

The all-parallel modes can be simulated by using the preserving strictly sequential mode, but obviously the simulation usually cannot be real-time. Therefore the input stream has to be adapted accordingly.

**Theorem 8.** *Every NCCAB over an infinite commutative monoid $G$ working in one of the rule application modes from $\{allP, pcallP\}$ can be simulated by an NCC working in the rule application mode pssequ, with the input stream being constant for a bounded number of steps.*

*Proof. (Sketch)* As in the proof of Theorem 2 we can use an additional control variable $z$ to guide the simulation of the recalculation of all other variables step by step applying all applicable functions. The maximal number of these sequences defines a constant $c$. The input stream for the input variables has to keep their values for a time period of $c$ steps. Moreover, the output variables get their corresponding correct values only at the end of each period of $c$ derivation steps. □

## 6 Conclusions

In this paper, we have introduced numerical networks of cells with forward activation and blocking of rules. The proposed mechanism separates the applicability of a rule, determined by its guard, from its availability, determined by the activation and blocking information generated during previous computation steps. This additional control layer makes dependencies between rule applications explicit and provides a natural way to organize computations into successive phases. The definition was given for the main rule application modes of numerical networks of cells, including the all-parallel and the sequential mode as well as their preserving variants, and in addtion for a strictly sequential evolution where at most one rule is applied in the overall system.

The examples and simulation results show that activation and blocking of rules can coordinate non-trivial numerical computations. Under suitable assumptions on the underlying numerical domain, activation and blocking can be encoded by auxiliary control variables and extended guards. Thus, the new mechanism does not extend the computational power of numerical networks of cells in these settings, but offers a clearer and more modular representation of computational control.

Several directions remain open for further investigation. These include identifying minimal activation and blocking structures for relevant classes of computations, studying the descriptional and temporal costs of translations between application modes, and examining restricted forms of guards and producer functions. In particular, it would be interesting to investigate the class of NNCABs with guards that are always true, or equivalently with no guards. This would allow to precisely show the power of the proposed control mechanism.

Most of the examples in this paper have only used activations of rules. Hence, it is a challenge for future research to find examples where blockings of rules can be used for more efficient simulations.

Another interesting idea is to consider the possibility of backward activation and blocking of rules, where the applicability of a rule depends on the activation and blocking information generated during future computation steps.

**Acknowledgements**

The authors gratefully acknowledge the careful reading and the useful comments of the referees. Artiom Alhazov acknowledges Project 011301 "Information systems based on artificial intelligence" by the Moldova State University.

## References

1. Alhazov, A., Freund, R., Ivanov, S.: Variants of P systems with activation and blocking of rules. Natural Computing **18**(3), 593–608 (2019). https://doi.org/10.1007/s11047-019-09747-5
2. Alhazov, A., Freund, R., Ivanov, S., Verlan, S.: Numerical networks of cells. In: 23rd Conference on Membrane Computing, 6-9 September 2022, Trieste, Italy (2022)
3. Buiu, C., Vasile, C.I., Arsene, O.: Development of membrane controllers for mobile robots. Information Sciences **187**, 33–51 (2012). https://doi.org/10.1016/j.ins.2011.10.007
4. Florea, A.G., Buiu, C.: Sensor fusion for autonomous drone waypoint navigation using ROS and numerical P systems: A critical analysis of its advantages and limitations. In: 2019 22nd International Conference on Control Systems and Computer Science (CSCS). pp. 112–117 (2019). https://doi.org/10.1109/CSCS.2019.00027
5. Freund, R., Pérez-Hurtado, I., Riscos-Núñez, A., Verlan, S.: A formalization of membrane systems with dynamically evolving structures. International Journal of Computer Mathematics **90**(4), 801–815 (2013). https://doi.org/10.1080/00207160.2012.748899
6. Freund, R., Verlan, S.: A formal framework for static (tissue) P systems. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 4860, pp. 271–284. Springer (2007). https://doi.org/10.1007/978-3-540-77312-2_17
7. Hopcroft, J., Pansiot, J.J.: On the reachability problem for 5-dimensional vector addition systems. Theoretical Computer Science **8**(2), 135–159 (1979)
8. Minsky, M.L.: Computation. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
9. Pavel, A., Arsene, O., Buiu, C.: Enzymatic numerical P systems – a new class of membrane computing systems. In: Fifth International Conference on Bio-Inspired Computing: Theories and Applications, BIC-TA 2010, Changsha, China, September 23-26, 2010. pp. 1331–1336. IEEE (2010). https://doi.org/10.1109/BICTA.2010.5645071

10. Păun, Gh., Păun, R.A.: Membrane computing and economics: Numerical P systems. Fundamenta Informaticae **73**(1-2), 213–227 (2006), http://content.iospress.com/articles/fundamenta-informaticae/fi73-1-2-20
11. Shang, Z., Verlan, S., Zhang, G., Rong, H.: FPGA implementation of numerical P systems. International Journal of Unconventional Computing **16**(2-3), 279–302 (2021), https://www.oldcitypublishing.com/journals/ijuc-home/ijuc-issue-contents/ijuc-volume-16-number-2-3-2021/ijuc-16-2-3-p-279-302/
12. Verlan, S., Freund, R., Alhazov, A., Ivanov, S., Pan, L.: A formal framework for spiking neural P systems. Journal of Membrane Computing **2**, 355–368 (2020). https://doi.org/10.1007/s41965-020-00050-2
13. Verlan, S., Zhang, G.: A tutorial on the formal framework for spiking neural P systems. Natural Computing (2022). https://doi.org/10.1007/s11047-022-09896-0
14. Wang, X., Zhang, G., Gou, X., Paul, P., Neri, F., Rong, H., Yang, Q., Zhang, H.: Multi-behaviors coordination controller design with enzymatic numerical P systems for robots. Integrated Computer-Aided Engineering **28**(2), i119–140 (2021). https://doi.org/10.3233/ICA-200627
15. Wang, X., Zhang, G., Neri, F., Jiang, T., Zhao, J., Gheorghe, M., Ipate, F., Lefticaru, R.: Design and implementation of membrane controllers for trajectory tracking of nonholonomic wheeled mobile robots. Integrated Computer-Aided Engineering **23**(1), 15–30 (2016). https://doi.org/10.3233/ICA-150503
16. Zhang, G., Pérez-Jiménez, M.J., Núñes, A.R., Verlan, S., Konur, S., Hinze, T., Gheorghe, M.: Membrane Computing Models: Implementations. Springer (2021), https://link.springer.com/book/10.1007/978-981-16-1566-5

# A matrix-based and GPU-accelerated simulator for ASNP systems

Matteo Balzerani[2 [0009−0004−7323−4307]], Sandro Erba[1 [0009−0001−3320−6166]], Giuditta Franco[2 [0000−0003−1447−5253]], and Claudio Zandron[1 [0000−0002−2163−7639]]

[1] Dept. of Informatics, Systems and Communication, University of Milano-Bicocca, Milan, Italy
[2] Dept. of Computer Science, University of Verona, Verona, Italy

**Abstract.** Spiking Neural P systems with anti-spikes (ASNP systems) are a variant of Spiking Neural P systems, inspired by the way biological neurons process information through both excitatory and inhibitory impulses. In this work, we propose a matrix representation for ASNP systems that explicitly accounts for anti-spikes and inhibitory synapses, by extending previously introduced matrix-based models for SN P systems. The proposed representation is designed to exploit the parallel architecture of GPUs, where neuron dynamics are mapped onto operations over sparse matrices. Based on this formalism, we implement and evaluate a GPU simulator by comparison against two other approaches: a CPU-based matrix implementation and an object-oriented implementation of the same system. As a case study, we consider a large-scale ASNP system for image recognition, whose size in terms of neurons, synapses, and rules substantially exceeds those employed in previous GPU-based simulators. Experimental results confirm that the GPU-based matrix simulator achieves significantly lower execution times and better scalability compared to the alternative implementations. Finally, directions for future work are outlined.

**Keywords:** membrane systems · extended spiking neural P networks · GPU based parallel computing

## 1 Introduction

Membrane computing was introduced by Gheorghe Păun [18] as a branch of natural computing inspired by the structure of biological cells, where objects evolve in parallel according to the application of rewriting rules associated to membrane compartments. For a general introduction to membrane computing and P systems, the reader is referred to [9], and to [11] for a comprehensive list of possible extensions to the model. Within this framework, Spiking Neural P systems [14] (SN P systems) were introduced as a biologically inspired model abstracting the behaviour of neurons that communicate through discrete electrical impulses, called spikes. In SN P systems, neurons are organized in a directed

graph and contain multisets of spikes that evolve according to firing and forgetting rules. Computation proceeds in a synchronous manner: at each step, all the enabled neurons fire simultaneously, so producing a highly parallel evolution of the system [15].

In [21], Zeng, Adorna, Martínez-del Amor, Pan, and Pérez-Jiménez showed that SN P systems may be naturally represented by matrices, thus allowing the implementation of an SN P system simulator employing parallel devices such as GPUs. A first attempt appeared in the early 2010s, when Cabarle et al. introduced a GPU-based simulator for SN P systems without delays [3]. In this initial work, the main challenge was identified as the efficient simulation of SN P systems, which was resolved through the matrix-based formalism. Shortly after, the initial simulator was refined [6, 5] and evaluated under different input configurations, to confirm the performance benefits of GPU acceleration. Subsequent work addressed the challenge of non-determinism [7], introducing algorithms capable of handling multiple computation paths. These developments are comprehensively reviewed in [22], which surveys GPU- and FPGA-based implementations, optimization strategies, and current research challenges.

Later, a sparse matrix-vector representation was introduced to improve memory efficiency and computational performance [1, 13]. These techniques proved particularly effective for large-scale SN P systems, and are also used in the simulations carried out in this work. More recently, GPU-based simulation has been extended to evolutionary SN P systems [12], where GPUs are used for both system simulation and for evolutionary operators such as mutation and crossover. Furthermore, a GPU-accelerated web-based simulator, GPUSnapse, has been developed and improved by optimizing sparse matrix operations, reducing memory usage [16]. The matrix-based formalism has also been extended to other variants of the model, such as numerical spiking neural P systems, for which a matrix representation has been validated on the subset sum problem [2].

Building upon prior matrix-based work, this paper introduces several extensions of models present in the literature, to simulate large-scale SN P systems efficiently. First, we consider a substantially larger application scenario based on image recognition, involving models whose size exceeds by several orders of magnitude those used in previous GPU-based studies. This required extending the formalism to support a continuous spike train delivered across multiple input neurons throughout the entire computation, with rule applicability evaluated at each step after the input is injected. Furthermore, we extend the underlying matrix representation to support anti-spikes and inhibitory synapses, requiring modifications to the involved vectors and matrices. Finally, we provide a comprehensive performance evaluation by comparing matrix-based GPU and CPU implementations, as well as an object-oriented implementation, which, to the best of the authors' knowledge, has not been considered before.

The remainder of this paper is organized as follows. In section 2 we define the proposed SN P system model with its extensions, whereas in Section 3 the vector and matrix-based formalism used to simulate the proposed model is introduced.

Our simulator and the experimental results are described in Section 4. Finally, Section 5 concludes the paper and outlines possible directions for future work.

## 2 SN P systems with Anti-spikes and Inhibitory synapses

As described in [19], a SN P system with Anti-spikes (ASNP system, shortly), of degree $m \geq 1$ is a construct of the form

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out),$$

where

- $O = \{a, \bar{a}\}$ is the alphabet of elements inside the neurons, where $a$ denotes a spike and $\bar{a}$ an anti-spike;
- $\sigma_1, \sigma_2, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i)$ with $i = 1, 2, \ldots, m$, where
  - $n_i \geq 0$ is the initial number of spikes contained in $\sigma_i$;
  - $R_i$ is a finite set of rules of the following two forms:
    1. $E/b^c \rightarrow b^p$, where $E$ is a regular expression over $\{a\}$ or over $\{\bar{a}\}$, $b \in O$, $c, p \in \mathbb{N}$ and $c \geq p \geq 1$;
    2. $b^v \rightarrow \lambda$, for some $v \geq 1$, with the restriction that for each rule $E/b^c \rightarrow b^p$ of type 1 from $R_i$, we have $b^v \notin L(E)$;
- $syn = syn_e \cup syn_i \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, with $(i, i) \notin syn$, is the set of synapses between neurons, where:
  - $syn_e$ is the set of **excitatory** synapses,
  - $syn_i$ is the set of **inhibitory** synapses.

  **Spikes transmitted through inhibitory synapses are transformed into anti-spikes** [17]. In a synapse, the neuron from which the signal originates is called the pre-synaptic neuron, whilst the receiving neuron is called the post-synaptic neuron;
- $in, out \subseteq \{1, 2, \ldots, m\}$ denote the sets of input and output neurons.

The rules of the form $E/b^c \rightarrow b^p$ are called **firing rules**. If neuron $\sigma_x$ contains $n_x$ spikes/anti-spikes with $b^{n_x} \in L(E)$ and $n_x \geq c$, then rule $E/b^c \rightarrow b^p$ can be applied. This means that $c$ spikes/anti-spikes are consumed (thus $n_x - c$ spikes/anti-spikes remain in neuron $\sigma_x$) and $p$ spikes/anti-spikes are sent to each neuron $\sigma_y$ such that $(x, y) \in syn$. Every neuron can contain several rules. Because two firing rules, $E_1/b^{c_1} \rightarrow b^{p_1}$ and $E_2/b^{c_2} \rightarrow b^{p_2}$, can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more spiking rules can be applied in a neuron at some moment, while only one of them is chosen non-deterministically.

For any spiking rule $E/b^c \rightarrow b^p$, if $L(E) = b^c$, then it is called **pure** and can be written as $b^c \rightarrow b^p$ for short.

If an anti-spike meets a spike in a given neuron, then they annihilate each other instantaneously with the (implicit) rule $a\bar{a} \rightarrow \lambda$.

The rules of the form $b^v \rightarrow \lambda$, with $v \geq 1$, are called **forgetting rules**. If neuron $\sigma_i$ contains exactly $v$ spikes/anti-spikes, the forgetting rule $b^v \rightarrow \lambda$

can be applied, removing $v$ spikes/anti-spikes from the neuron. Given a firing rule $E/b^c \rightarrow b'^p$ if we consider $b$ as the left-hand side rule symbol and $b'$ as the right-hand side rule symbol, as explained in [17] we have four categories of rules identified by $(b, b') \in \{(a, a), (a, \overline{a}), (\overline{a}, a), (\overline{a}, \overline{a})\}$

The application of rules within neurons, executed in parallel and in a synchronised manner, enables the system to change its configuration. The **configuration** of ASNP systems is of the form $(n_1, n_2, \ldots, n_m)$ with $n_i \in \mathbb{Z}$. At any moment, if $n_i > 0$, it means that there are $n_i$ spikes in neuron $\sigma_i$; if $n_i < 0$, it indicates that neuron $\sigma_i$ contains $n_i$ anti-spikes. By using firing and forgetting rules, one can define a **transition** as a set of at most $n$ rules which are applied to step from one configuration to the next one. A series of transitions starting from the initial configuration is called **computation**. The computation halts if it proceeds to a configuration such that no rules can be applied in any neuron.

According to the original formulation [17], when inhibitory synapses are employed, anti-spikes are generated exclusively through them. Consequently, firing rules producing anti-spikes are excluded from this formulation and therefore the $b' = \overline{a}$ case cannot occur. In [19] it has been proved that ASNP systems with these constraints are universal. The matrix representation introduced in Section 3 is meant to be more general, allowing the encoding of both systems with explicit anti-spike production and systems with inhibitory synapses. Therefore, the enforcement of this modelling constraint is left to the implementation of the specific SN P system under consideration.

Earlier GPU simulators adopted only the rule representation based on pure firing rules. In work [4] the authors extended the original framework by introducing support for a broader class of firing rules based on regular expressions, while in [8] they further generalized the approach to handle all commonly used rule forms in SN P systems. This increase in expressiveness, however, came at a computational cost: rule applicability checks, performed in constant time for pure rules, required linear-time evaluation in the generalized framework.

Since the focus of this work is the efficient simulation of large-scale SN P systems, only pure rules without delays are considered. This restriction allows rule applicability to be determined through simple numerical comparisons, by avoiding the evaluation of regular expressions $E$. As a consequence, all computations can be expressed in matrix form and executed entirely on the GPU, without requiring intermediate transfers to the CPU for rule applicability checks. This problem could be overcome as described in [13], where the use of a specific kernel function extends the range of regular expressions that can be applied.

Furthermore, the proposed model is deterministic. In each neuron, pure rules are defined for distinct charge values, so ensuring that there is no couple of rules simultaneously applicable. Therefore, the non-determinism arising from overlapping rule languages, *i.e.* the condition $L(E_1) \cap L(E_2) \neq \emptyset$, does not occur. Under this assumption, at most one rule can be applied in each neuron at every computation step.

## 3 Matrix Representation of an ASNP system

In [21] it is shown how the computation of SN P systems can be represented by operations with matrices. In this section, that matrix representation of SN P systems is extended with the implementation of anti-spikes, along with an example illustrating the introduced definitions.

Let $\Pi$ be a deterministic ASNP system with $m$ neurons and $n$ pure rules. We use the following definitions to represent the evolution of $\Pi$ configurations. Note that definitions 1, 2 and 3 depend on the computational step $k$, and therefore define concepts varying (and to be computed) at each time step, whilst definitions 4, 5, 6 and 7 are computed at the start of the simulation and are constant.

**Definition 1 (Configuration vector $C_k$).** For any $k \in \mathbb{N}$, the vector $C_k = (n_1^{(k)}, n_2^{(k)}, \ldots, n_m^{(k)})$ is called the $k$-th configuration vector of the system, where $n_i^{(k)}$ is an integer number denoting the amount of spikes or anti-spikes in neuron $\sigma_i$, $i = 1, 2, \ldots, m$, at the $k$-th step of the computation. A positive value represents spikes, while a negative one represents anti-spike, so this integer denotes the charge value of each neuron.

In order to describe which rules are chosen and applied in each configuration, the spiking vector is defined.

**Definition 2 (Spiking vector $S^{(k)}$).** Given a total order over the rules in $\Pi$, they may be referred to accordingly, as $r_1, r_2, \ldots, r_n$. A spiking vector $S^{(k)}$ is defined as: $S^{(k)} = (r_1^{(k)}, r_2^{(k)}, \ldots, r_n^{(k)})$, where:

$$r_j^{(k)} = \begin{cases} 1, & \text{if } E_j \text{ of rule } r_j \text{ at the } k\text{-th step is satisfied;} \\ 0, & \text{otherwise.} \end{cases}$$

Since the proposed model is deterministic, if $E_j$ is satisfied then the corresponding rule $r_j$ is applied consequently. In order to establish whether $E_j$ is satisfied, we define the following two vectors.

**Definition 3 (Extended configuration vector $eC_k$).** The extended configuration vector of $\Pi$ consists of the charge value of each neuron, repeated a number of times equal to the number of rules in that neuron: $eC_k = (n_{1,1}^{(k)}, \ldots, n_{m,n}^{(k)})$.

Let us note that the configuration vector $C_k$ has dimension $m$ (number of neurons), while the extended configuration vector $eC_k$ has $n$ components (total number of rules).

**Definition 4 (Rule vector $rV$).** With all rules in the form $a^c \to a^p$, the rule vector $rV$ is defined as a vector consisting of the value $c$ for each rule:

$$rV = (c_1, c_2, \ldots c_n)$$

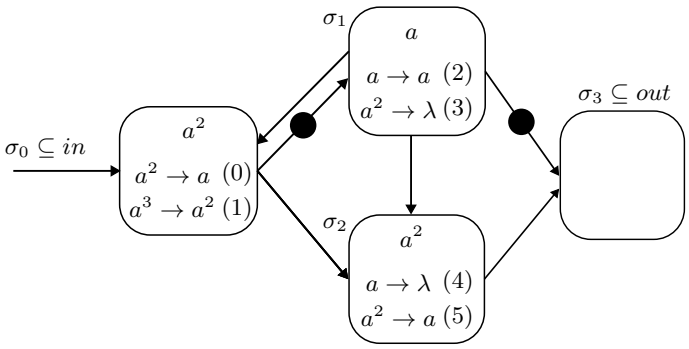



Fig. 1: Example of an ASNP system $\Pi$

**Example:** Let $\Pi$ be the following ASNP system, for $k = 0$:
On the setting reported in Figure 1 we can calculate the following vectors:

$$C_0 = (2, 1, 2, 0) \qquad S^{(0)} = (1, 0, 1, 0, 0, 1)$$
$$eC_0 = (2, 2, 1, 1, 2, 2) \qquad rV = (2, 3, 1, 2, 1, 2)$$

Intuitively, the applicability condition for a pure rule reduces to verifying the identity $E = b^c$. Under the assumption (by definition) that $c \geq p$, it follows that the spiking vector may be computed exclusively by means of the following equality:

$$S^{(k)} = \lfloor oneVec \ \oslash (oneVec + |eC_k - rV|) \rfloor \tag{1}$$

where *oneVector* is a row-vector composed of $n$-ones and $\oslash$ represents the Hadamard (element-wise) division of the two vectors. Applying Equation (1) iteratively, the evolution of the system in the example is:

$$C_0 = (2, 1, 2, 0), \qquad S^{(0)} = (1, 0, 1, 0, 0, 1),$$
$$C_1 = (1, -1, 2, 0), \qquad S^{(1)} = (0, 0, 0, 0, 0, 1),$$
$$C_2 = (1, -1, 0, 1), \qquad S^{(2)} = (0, 0, 0, 0, 0, 0).$$

Since in $S^{(2)}$, no rule is applicable, the computation halts.

This formulation can be positioned within the general framework for spiking neural P systems introduced by Verlan et al. [20]. In particular, the signed configuration vectors used in ASNP systems naturally correspond to integer vectors, consistently with the treatment of anti-spikes in that framework, where spike–anti-spike annihilation is represented by the cancellation of positive and negative values.

In the following, we define the spiking transition matrix to report how the rules affect the charges present in the P system, and the synapses matrix to represent the synaptic connectivity associated with each rule. This matrix identifies the neurons from which spikes are emitted and the neurons to which they are transmitted.

**Definition 5 (Spiking transition matrix $M_\Pi$).** Let the spiking transition matrix $M_\Pi = [a_{ji}]_{n\times m}$, where:

$$a_{ji} = \begin{cases} -c, & \text{if rule } r_j \text{ is in neuron } \sigma_i \text{ and it is applied consuming } c \text{ spikes;} \\ +c, & \text{if rule } r_j \text{ is in neuron } \sigma_i \text{ and it is applied consuming } c \text{ anti-spikes;} \\ -p, & \text{if rule } r_j \text{ is in neuron } \sigma_s,\ s \neq i\ ,\ (s,i) \in syn, \\ & \text{and it is applied producing } p \text{ anti-spikes;} \\ p, & \text{if rule } r_j \text{ is in neuron } \sigma_s,\ s \neq i\ ,\ (s,i) \in syn, \\ & \text{and it is applied producing } p \text{ spikes;} \\ 0, & \text{otherwise.} \end{cases}$$

**Definition 6 (Synapses matrix $syn_{M_\Pi}$).** Let the synapses matrix $syn_{M_\Pi} = [se_{ji}]_{n\times m}$, where:

$$se_{ji} = \begin{cases} 1, & \text{if rule } r_j \text{ is in the pre-synaptic neuron } \sigma_i \textbf{ or} \\ & \text{rule } r_j \text{ is in neuron } \sigma_s,\ s \neq i\ ,\ (s,i) \in syn_e; \\ -1, & \text{if rule } r_j \text{ is in neuron } \sigma_s,\ s \neq i\ ,\ (s,i) \in syn_i; \\ 0, & \text{otherwise.} \end{cases}$$

Let us clarify that $M_\Pi$ only encodes the amount of spikes/anti-spikes produced or consumed by each rule, without including any information about inhibitory synapses. Conversely, $syn_{M_\Pi}$ incorporates the synaptic structure and the corresponding sign transformation. Therefore, the Hadamard product between $M_\Pi$ and $syn_{M_\Pi}$ applies the inhibitory effect, producing the final $n \times m$-dimensional matrix reporting the spikes/anti-spikes transmitted through the synapses.

We note that $syn_{M_\Pi}$ contains duplicate information because, for each rule present in the same neuron, the rows are identical, as they represent the same synaptic structure: however, this is required to realize component-wise multiplication.

**Definition 7 (Synapses spiking transition matrix $sM_\Pi$).** The synapses spiking transition matrix represents the transition structure of each rule enriched with synaptic information, and is defined as:

$$sM_\Pi = M_\Pi \odot syn_{M_\Pi}$$

where $\odot$ denotes the Hadamard product of the two matrices, such that: $sM_\Pi = [a_{ji} \cdot se_{ji}]_{n\times m}$

For example, based on the setting reported in Figure 1, matrices from previous definitions are computed as:

$$M_{\Pi} = \begin{pmatrix} -2 & 1 & 1 & 0 \\ -3 & 2 & 2 & 0 \\ 1 & -1 & 1 & 1 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -2 & 1 \end{pmatrix} \quad syn_{M_{\Pi}} = \begin{pmatrix} 1 & -1 & 1 & 0 \\ 1 & -1 & 1 & 0 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad sM_{\Pi} = \begin{pmatrix} -2 & -1 & 1 & 0 \\ -3 & -2 & 2 & 0 \\ 1 & -1 & 1 & -1 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -2 & 1 \end{pmatrix}$$

Given the charge configuration vector at time $k$ ($C_k$), the spiking vector specifying the rules applied at time $k$ ($S^{(k)}$), and the matrix that shows the effect these rules have on the charges in the $\Pi$ system ($sM_{\Pi}$), it is possible to define the next configuration recursively as

$$C_{k+1} = C_k + S^{(k)} \cdot sM_{\Pi} \tag{2}$$

Equivalently, the next configuration can be written as:

$$C_{k+1} = C_k + \lfloor oneVec \oslash (oneVec + |eC_k - rV|) \rfloor \cdot (M_{\Pi} \odot syn_{M_{\Pi}}) \tag{3}$$

In pseudocode, the above equation is implemented by the following algorithm:

**Algorithm 1** Matrix-based ASNP simulation step

**Require:** $C_k \in \mathbb{Z}^m$, $eC_k \in \mathbb{Z}^n$, $rV \in \mathbb{Z}^n$, $M_{\Pi} \in \mathbb{Z}^{n \times m}$, $\text{syn}_{M_{\Pi}} \in \mathbb{Z}^{n \times m}$ and $I_k \in \mathbb{Z}^m$ as the input spike-train injected at given time $k$.
**Ensure:** $C_{k+1} \in \mathbb{Z}^m$

$C_k \leftarrow C_k + I_k$ ▷ Input injection
$S^{(k)} \leftarrow \lfloor \mathbf{1} \oslash (\mathbf{1} + |eC_k - rV|) \rfloor$ ▷ Spiking vector as per Definition 2
$sM_{\Pi} \leftarrow M_{\Pi} \odot \text{syn}_{M_{\Pi}}$ ▷ Precompute synapses spiking transition matrix
$\Delta C \leftarrow S^{(k)} \cdot sM_{\Pi}$ ▷ Matrix-vector multiplication as per Eq. (2)
$C_{k+1} \leftarrow C_k + \Delta C$ ▷ Configuration update as per Eq. (2)
**return** $C_{k+1}$

## 4 Results

To evaluate the proposed matrix representation, we consider the ASNP system introduced in [10] for an image classification task. The model processes the handwritten digit MNIST dataset, where each $28 \times 28$ grayscale image is encoded as a spike train and sequentially injected into the input neurons. The network produces one of the ten output classes corresponding to the digits from 0 to 9. Although the structural details of the ASNP network are beyond the scope of this paper and can be found in [10], a classification requires four computation steps. Images are processed sequentially, so the simulation of $x$ images requires

$x + 3$ computation steps to fully propagate the last input through the network. The simulation terminates once the predefined number of input images has been processed.

Before being injected into the network, each image undergoes a preprocessing stage in which pixel intensities are quantized according to a user-defined quantization range. The second layer performs convolution using multiple kernels, while the third layer applies an average pooling operation to reduce the resulting feature maps. Finally, the synaptic weights connecting the last two layers are obtained through an offline training procedure based on external classifiers, namely Support Vector Machines (SVM) and Logistic Regression (LR), whose learned parameters are subsequently converted into an equivalent SN P system representation.

The matrix-based implementation of this model exhibits all the features discussed in the previous sections, including anti-spikes, inhibitory synapses, pure and deterministic rules. All implementations considered in this work employ the same input encoding, preprocessing procedure, halting criterion, and numerical representation to ensure a fair comparison. The source code of the simulator is publicly available online [3]

The selected model was chosen because an Object-Oriented Programming (OOP) implementation is available. Starting from this implementation, the ASNP system was translated into the matrix formalism introduced in the previous sections. This allows a direct comparison between different execution strategies while preserving exactly the same underlying P system.

The experimental analysis compares three implementations of the same ASNP system:

- the original OOP implementation, for short **OOPSNPS**;
- the matrix implementation executed on a GPU, for short **MSNPS(GPU)**;
- the matrix implementation executed on a CPU, for short **MSNPS(CPU)**.

In order to execute the image classification system, we used a desktop workstation running Ubuntu Linux, with a 13th Gen Intel Core i9-13900KF CPU, 24 cores (32 threads), 6.00 GHz, and with a 36MB L3 cache having 62GB of RAM. The GPU of the system is an NVIDIA GeForce RTX 4090 with 24GB Video RAM, 16,384 CUDA cores, running CUDA version 12.6.

OOPSNPS and MSNP(CPU) runs solely on the CPU, while MSNPS(GPU) runs on both the CPU and the GPU. As in [3], Python was chosen as the implementation language. In our case, though, we extended this by using PyTorch (v2.12.0) for GPU/CUDA acceleration.

A key advantage of PyTorch is its device-agnostic matrix API: the same code runs on CPU or GPU simply by choosing the target device, meaning that MSNPS(CPU) and MSNPS(GPU) share all implementation details except the device itself.[4]

[3] Available at: https://github.com/MatteoBalzerani/matrix-spiking-neural-p-system (accessed: June 2026).

[4] For documentation about PyTorch's device parameter, see https://docs.pytorch.org/cppdocs/api/c10/device.html#c10-device-class.

Since all implementations execute the same ASNP system, their outputs are expected to be consistent for the considered dataset. In our experiments, all implementations produce identical classification results on the same input data. Therefore, classification performance metrics are not reported, as the purpose of this comparison is not to assess classification accuracy, but rather to evaluate the computational efficiency of the different implementations.

GPU timings were obtained using timed CUDA events with `torch.cuda.synchronize()` before and after each measurement, ensuring a clean pipeline and reliable `elapsed_time()` readings. This isolates kernel execution time, excluding CPU overhead and host-device transfers. No GPU warm-up was performed, and timings exclude data transfers and matrix construction. These measurements capture the cost of evaluating the transition in Eq. (3) at each step. [5]

As anticipated in the introductory section, only inhibitory synapses were adopted (i.e., with no anti-spike-producing firing rules) along with sparse matrix-vector representations, to improve memory efficiency and computational performance. Sparsity was handled using PyTorch's CSR [6] representation.

The considered model exposes several configurable parameters affecting the size of the system. In particular, variations in these parameters modify the total number of neurons, rules and synapses involved in the computation. The reported experiments analyse how the execution time changes as these parameters vary, highlighting the scalability characteristics of the proposed matrix-based implementation. For each set of parameters, each system has been repeatedly executed three times (seeds 42, 999, 1234) and the results were averaged.

The random seed does not affect the execution time of the simulator, as the considered computation is deterministic. The observed variability among different runs is instead due to the different execution times of individual computation steps (e.g., processing different input images), whose costs may slightly vary during the simulation. Therefore, multiple runs with different seeds are used only to mitigate runtime fluctuations and obtain more stable average measurements.

Firstly, with a fixed number of 8,906 neurons, we analysed the execution times of each system by varying the number of rules from approximately ∼582K to ∼978K. From the results obtained, in particular by the magnitude of the standard deviation, it can be observed that variations in the number of rules leave the average execution times essentially unaffected. Hence, the histogram presented in Figure 2 comprises just three data points, each representing the average time (in milliseconds) taken by each system to execute a single step. On average, MSNPS(GPU) is 6.66× faster than OOPSNPS and 43.92× faster than MSNPS(CPU).

Secondly, we scaled the number of neurons, with the number of rules increasing proportionally. As illustrated in Figure 3, OOPSNPS and MSNPS(GPU)

[5] See https://docs.pytorch.org/docs/2.12/generated/torch.cuda.Event.html and https://docs.pytorch.org/docs/2.12/generated/torch.cuda.synchronize.html

[6] Further details on PyTorch's CSR representation, advantages and operations can be found at https://docs.pytorch.org/docs/2.12/sparse.html#sparse-csr-docs

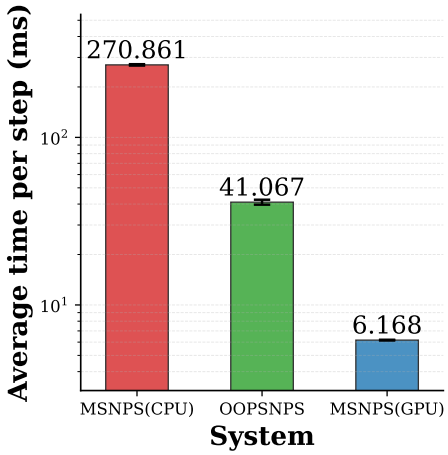



Fig. 2: Average time per step across the three systems.

scale linearly, with MSNPS(GPU) remaining consistently faster, whereas MSNPS(CPU) shows a markedly steeper increase in execution time.

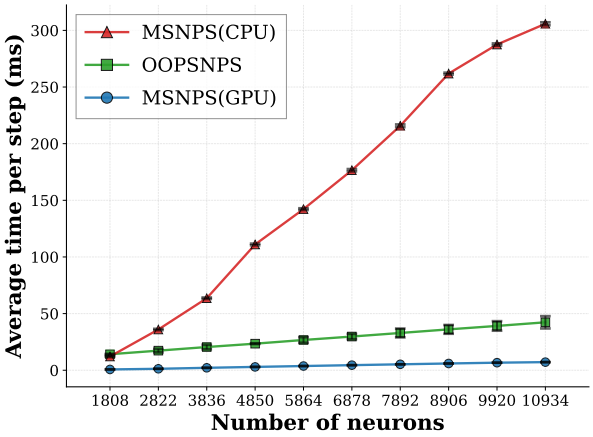



Fig. 3: Execution times with varying number of neurons.

Finally, to better understand the execution time differences observed in the previous experiments, we investigated the impact of model sparsity on execution time. We hypothesize that this advantage stems from the highly sparse connectivity of the network, where each neuron is connected only to a limited number of neurons in the immediately following layer, with no connections to neurons in the same layer, preceding layers, or layers beyond the next one. This topol-

ogy results in extremely sparse synaptic matrices, favoring the object-oriented implementation by restricting computation to active neurons and their outgoing connections, whereas the matrix-based approach is designed to account for the entire connectivity structure.

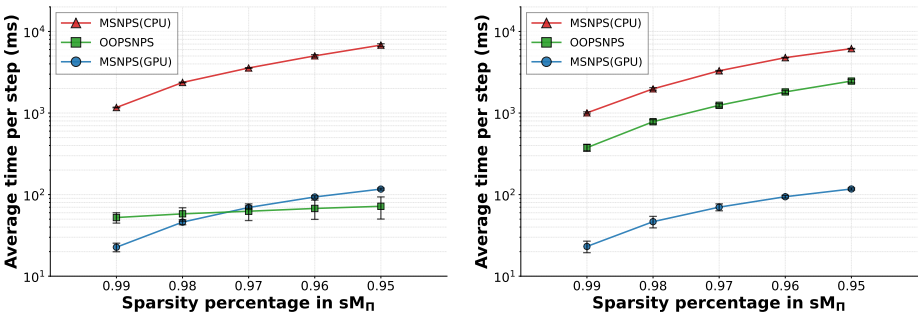
Specifically, we varied the proportion of zero-valued entries in the matrix $sM_\Pi$, which, as shown in Equation 2, is applied at every simulation step.

The model used in the experiments so far exhibits an inherent sparsity of approximately 99.8%. To evaluate the effect of this characteristic, the sparsity was artificially reduced by up to 95%, and the experiments were conducted under two different simulation settings.

In the first scenario, shown in Figure 4a, firing rules are only rarely enabled, resulting in a limited number of transmitted spikes. This configuration favours the object-oriented implementation, as fewer spike transmission and reception operations are required during the simulation.

In the second scenario, shown in Figure 4b, the charge configuration vector was adjusted at every simulation step to ensure that at least one firing rule per neuron was always applicable. Consequently, every neuron emits spikes at every step, substantially increasing the computational workload of the OOP simulator. In contrast, the execution time of the matrix-based SNP system implementation remains essentially unaffected by the number of firing rules that are activated, since its computations are performed through fixed matrix operations regardless of the network activity.

In general, the results clearly show that sparsity has a significant impact on computational performance. As the sparsity decreases, the execution time of the matrix-based implementation increases markedly in both scenarios, highlighting the importance of sparse matrix representations for achieving efficient GPU-based simulations.




(a) Execution times with varying charges. (b) Execution times with fixed charges

Fig. 4: Execution times with varying sparsity of the $sM_\Pi$ matrix.

## 5 Conclusions

This paper introduced a matrix representation of ASNP systems that explicitly accounts for anti-spikes, inhibitory synapses, and images as continuous input spike trains, extending previous formalisms with three key contributions: *i*) support for inhibitory synapses and anti-spike dynamics, achieved through extensions of the underlying vector and matrix definitions; *ii*) a configuration vector handling multiple input and output neurons, enabling the injection of spike trains across the entire computation; and *iii*) a comprehensive performance comparison among GPU-based matrix, CPU-based matrix, and object-oriented implementations. The formalism was designed to exploit GPU parallelism via sparse linear algebra operations, and was evaluated on a large-scale image recognition system.

Experimental results confirmed that the GPU-based simulator achieves significantly lower execution times and better scalability, with performance remaining stable as the number of rules increases. We also verified that larger SNP systems can capitalize on the massively parallel architecture of GPUs compared to multi-core CPUs, and thus increase the speedup even further [4] than the previous obtained results. In fact, on average the SN P system on GPU is $6.66\times$ faster than the OOP one. Furthermore, we evaluated the impact of the model sparsity on performance. The results showed that execution time increases as sparsity decreases, confirming that the efficiency of the GPU-based matrix implementation strongly benefits from highly sparse matrix representations.

The main limitation of our extended model is its restriction to pure, deterministic rules without delays, while a natural further extension could include support for all the forms of regular-expression-based rules.

Another important research direction concerns the support of nondeterministic ASNP systems. Unlike deterministic models, multiple rules may be simultaneously applicable within the same neuron. While a straightforward implementation could randomly select one applicable rule, a faithful simulation should instead explore every admissible choice by generating independent computational branches that evolve in parallel. Efficiently managing the potentially exponential growth of such branches on massively parallel architectures represents a significant algorithmic challenge.

From the performance perspective, future work should investigate the behavior of the proposed framework on ASNP systems with denser synaptic topologies, providing a more comprehensive assessment of its scalability and computational performance under different connectivity patterns.

The introduction of GPU-compatible ASNP systems also opens opportunities for applications, that naturally benefit from the representation of both positive and negative values. One of the applications is neuromorphic computing, where energy-efficient computation is a primary objective, together with the development of bio-inspired machine learning architectures that explicitly exploit inhibitory interactions.

Finally, the matrix-based formulation advanced in this work, by starting from the seminal idea in [21], suggests a broader research perspective. An interesting question is whether the same methodology can be extended to other variants of

SN P systems, enabling efficient GPU implementations for an even wider class of membrane computing models, and progressively bringing the SN P systems framework closer to high-performance computing.

In conclusion, this work brings ASNP systems, previously lacking a GPU-compatible simulation framework, closer to high-performance computing, and makes computationally tractable large-scale tasks such as image recognition in the SN P systems framework.

## References

1. Martínez-del Amor, M.Á., Orellana-Martín, D., Pérez-Hurtado, I., Cabarle, F.G.C., Adorna, H.N.: Simulation of spiking neural P systems with sparse matrix-vector operations. Processes **9**(4), 690 (2021). https://doi.org/10.3390/pr9040690
2. Ballesteros, K.J., Cailipan, D.P.P., de la Cruz, R.T.A., Cabarle, F.G.C., Adorna, H.N.: Matrix representation and simulation algorithm of numerical spiking neural P systems. Journal of Membrane Computing **4**, 41–55 (2022). https://doi.org/10.1007/s41965-022-00093-7
3. Cabarle, F., Adorna, H., Martínez-del Amor, M.: Simulating spiking neural P systems without delays using GPUs. International Journal of Natural Computing Research **2**(2), 19–31 (2011). https://doi.org/10.4018/jncr.2011040102
4. Cabarle, F., Adorna, H., Martınez-del Amor, M.: A spiking neural P system simulator based on CUDA. lncs 7184 (2012). https://doi.org/10.1007/978-3-642-28024-5_8
5. Cabarle, F.G., Adorna, H., Martínez-del Amor, M.A., Pérez-Jiménez, M.J.: Spiking neural P system simulations on a high performance GPU platform. In: International Conference on Algorithms and Architectures for Parallel Processing. pp. 99–108. Springer (2011). https://doi.org/10.1007/978-3-642-24669-2_10
6. Cabarle, F.G.C., Adorna, H., Martinez-del Amor, M.A.: An improved GPU simulator for spiking neural P systems. In: 2011 Sixth International Conference on Bio-Inspired Computing: Theories and Applications. pp. 262–267. IEEE (2011). https://doi.org/10.1109/BIC-TA.2011.37
7. Carandang, J.P., Cabarle, F.G.C., Adorna, H.N., Hernandez, N.H.S., Martínez-del Amor, M.Á.: Handling non-determinism in spiking neural P systems: Algorithms and simulations. Fundamenta Informaticae **164**(2-3), 139–155 (2019). https://doi.org/10.3233/FI-2019-1759
8. Carandang, J.P., Villaflores, J.M.B., Cabarle, F.G.C., Adorna, H.N., Martinez-del Amor, M.A.: CuSNP: Spiking neural P systems simulators in CUDA. Romanian Journal of Information Science and Technology **20**(1), 57–70 (2017), https://www.romjist.ro/full-texts/paper550.pdf
9. Ciobanu, G., Păun, G., Pérez-Jiménez, M.J.: Applications of membrane computing, vol. 17. Springer (2006). https://doi.org/10.1007/3-540-29937-8
10. Erba, S., Franco, G., Zandron, C.: Toward recognizing medical imaging with a lightweight spiking neural P system. In: LNCS volume for Proceedings of UCNC2026. p. to appear. Springer (2026)
11. Gheorghe, M., Ipate, F., Kannan, K., Konur, S., Kuppusamy, L., Lefticaru, R., Mahendran, A., Niculescu, M.I.: Spiking neural P systems and kernel P systems. Journal of Membrane Computing pp. 1–23 (2025). https://doi.org/10.1007/s41965-025-00204-0

12. Gungon, R.V., Hernandez, K.K.M., Cabarle, F.G.C., De la Cruz, R.T.A., Adorna, H.N., Martínez-del Amor, M.Á., Orellana-Martín, D., Pérez-Hurtado, I.: GPU implementation of evolving spiking neural P systems. Neurocomputing **503**, 140–161 (2022). https://doi.org/10.1016/j.neucom.2022.06.094
13. Hernández-Tello, J., Martínez-del Amor, M.Á., Orellana-Martín, D., Cabarle, F.G.C.: Sparse spiking neural-like membrane systems on graphics processing units. International Journal of Neural Systems **34**(07), 2450038 (2024). https://doi.org/10.1142/S0129065724500382
14. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae **71**(2-3), 279–308 (2006). https://doi.org/10.3233/FUN-2006-712-308
15. Leporati, A., Mauri, G., Zandron, C.: Spiking neural P systems: main ideas and results. Natural Computing **21**(4), 629–649 (2022). https://doi.org/10.1007/s11047-022-09917-y
16. Odasco, A.N.L., Rey, M.L.M., Cabarle, F.G.C.: Improving GPU web simulations of spiking neural P systems. Journal of Membrane Computing **5**(4), 205–220 (2023). https://doi.org/10.1007/s41965-023-00128-7
17. Pan, L., P, G.: Spiking neural P systems with anti-spikes. International Journal of Computers, Communications and Control **4** (09 2009). https://doi.org/10.15837/ijccc.2009.3.2435
18. Păun, G.: Computing with membranes. Journal of computer and system sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.169
19. Song, T., Pan, L., Wang, J., Venkat, I., Subramanian, K., Abdullah, R.: Normal forms of spiking neural P systems with anti-spikes. IEEE transactions on nanobioscience **11** (09 2012). https://doi.org/10.1109/TNB.2012.2208122
20. Verlan, S., Freund, R., Alhazov, A., Ivanov, S., Pan, L.: A formal framework for spiking neural P systems. Journal of Membrane Computing **2**(4), 355–368 (2020). https://doi.org/10.1007/s41965-020-00050-2
21. Zeng, X., Adorna, H., Martínez-del Amor, M.Á., Pan, L., Pérez-Jiménez, M.J.: Matrix representation of spiking neural P systems. In: International conference on membrane computing. pp. 377–391. Springer (2010). https://doi.org/10.1007/978-3-642-18123-8_29
22. Zhang, G., Verlan, S., Wu, T., Cabarle, F.G.C., Xue, J., Orellana-Martín, D., Dong, J., Valencia-Cabrera, L., Pérez-Jiménez, M.J.: Hardware simulations of spiking neural P systems. In: Spiking Neural P Systems: Theory, Applications and Implementations, pp. 377–422. Springer (2024). https://doi.org/10.1007/978-981-97-9282-5_12

# Reaction systems with rough sets

Péter Battyányi[0000−0001−6703−9661]

University of Debrecen, Faculty of Informatics, Kassai út 26., Debrecen, H-4028, Hungary
battyanyi.peter@inf.unideb.hu

**Abstract.** This paper investigates rough reaction systems, that is, reaction systems equipped with approximation mechanisms based on the partition induced by a given equivalence relation. Building on the notions of lower and upper approximations, we introduce the concepts of definite and possible outcomes for reaction systems in situations where only approximate information is available, either about the underlying set or about the components of the reactions. We investigate several scenarios, depending on which parts of the system are only approximately known. We show that, in most cases, the predicted outcomes obtained from these approximations may deviate significantly from the actual evolution of the reaction system. On the other hand, we identify situations in which the evolution can be computed exactly despite the incomplete information. Finally, we pose the question of under which conditions these approximations behave as expected, giving lower and upper bounds on the evolution of the reaction system even after several computational steps.

**Keywords:** reaction system, rough set, upper and lower approximation, reaction system function

## 1 Introduction

Reaction systems were introduced by Ehrenfeucht, A. and Rozenberg, G. [13] to model biochemical processes in living cells. Their constituents are a finite set of entities and a finite set of reactions acting on them. A reaction $a$ is a triple $(R_a, I_a, P_a)$, where $R_a$ is the set of reactants, $I_a$ the set of inhibitors, and $P_a$ the set of products.

The computation of a reaction system is defined as a sequence of steps. At each step, the set of enabled reactions is determined by the current state, and the next state is obtained as the union of the products of all enabled reactions. A reaction is enabled in a state if all its reactants are present and none of its inhibitors occur. The state at each step is formed from entities supplied by the environment together with those produced in the previous step.

Since their introduction, reaction systems have been established as a general computational model with a wide range of applications in biological modelling, unconventional computational models, and theoretical computer science [2, 5, 7, 12, 8].

Rough sets, as introduced by Pawlak [15], were developed as a tool for handling approximate data and for analysing computational processes in the presence of partial information and incomplete knowledge. Rough set theory enjoys a wide range of applicability, covering both theoretical and practical domains [1, 10, 3, 16, 18].

Independently of the present work, the interplay between rough sets and reaction systems has already been explored in a few pioneering papers [6, 9]. In [9], a decision table is introduced to determine whether certain attributes hold for elements of the underlying set of a reaction system, and this table is then approximated from below and from above in the standard rough-set framework. The authors of [6] focus on the perception of the elements of a reaction system through tests, using these intrinsic tests to simplify the evaluation of the system's dynamics. Our approach differs fundamentally from those works. Rather than approximating auxiliary structures or exploiting tests, we investigate how an approximation space induced by an equivalence relation affects the theoretical properties of reaction systems themselves. This line of research has already been initiated in our previous work [4], where we examined the impact of an approximation space on the termination behavior of a reaction system and on the effectiveness of predicting the values of functions computed by the system. An approximation space consists of an arbitrary family of non-empty subsets of the underlying set, called the base sets.

In the present paper, we assume that an equivalence relation is given on the underlying set of the reaction system. As usual, the partition induced by this equivalence relation gives rise to the corresponding lower and upper approximations. Sets that are unions of equivalence classes are regarded as fully observable (definable), whereas all other sets are considered non-definable. Our main objective is to determine how accurately the evolution of a reaction system can be approximated when only partial information about either the underlying set or the reactions is available. To this end, we investigate several scenarios: neither the reactions nor the underlying set are definable, the reactions are core-definable (reactant and inhibitor sets are both definable) while the underlying set is non-definable, and the reactions are product-definable (the product set is definable) and the underlying set are definable. We show that, in general, the approximated evolution may differ substantially from the actual one. Nevertheless, in the latter two cases, one of the proposed approximations turns out to be exact. Finally, we briefly discuss conditions under which the exact evolution remains enclosed between the lower and upper approximations at every computational step. A complete characterization of such conditions is left for future work.

## 2 Definitions and notation

This section introduces the fundamental definitions and concepts used throughout the paper.

**Definition 1.** *[Reaction system]*

1. *A reaction is an ordered triplet of finite sets*

$$a = (R_a, I_a, P_a),$$

*such that* $R_a \cap I_a = \emptyset$ *and* $P_a, R_a$ *are nonempty, while* $I_a$ *can be empty. If* $R_a, I_a, P_a \subseteq S$ *for a set* $S$*, then we say that* $a$ *is a reaction in* $S$*.*
*The sets* $R_a$*,* $I_a$*, and* $P_a$ *are called the reactant set of* $a$*, the inhibitor set of* $a$*, and the product set of* $a$*, respectively. The sets* $R_a$*,* $I_a$ *together are called the core of* $a$*.*
2. *Let* $\mathcal{A} = (S, A)$ *be a reaction system in* $S$*. Let* $T \subseteq S$ *and* $a \in A$*. We say that* $a$ *is* enabled *in* $T$*, in notation* $en_a(T)$*, if* $R_a \subseteq T$ *and* $I_a \cap T = \emptyset$ *holds. We define the result of the application of* $a$ *to* $T$ *as follows:*

$$res_a(T) = \begin{cases} R_a & \text{if } en_a(T), \\ \emptyset & \text{otherwise.} \end{cases}$$

*The* effect *of* $\mathcal{A}$ *on* $T$ *is defined as*

$$res_{\mathcal{A}}(T) = \bigcup_{\{a \in A \mid en_a(T)\}} res_a(T).$$

To facilitate the presentation, we allow the inhibitor set to be empty, although all the results remain valid without this assumption. We recall the following example from [12], which illustrates the evolution of a reaction system.

*Example 1.* Consider the reaction system

$$\mathcal{U} = (\{1, 2, 3, 4\}, U),$$

where the set $U$ consists of the following reactions:

$$(\{1\}, \{3\}, \{2\}),$$
$$(\{2\}, \{1\}, \{1\}),$$
$$(\{2\}, \{3\}, \{3\}),$$
$$(\{3\}, \{1, 2\}, \{1, 2, 4\}),$$
$$(\{4\}, \{3\}, \{1, 2\}),$$
$$(\{1, 3\}, \{2, 4\}, \{2, 3\}).$$

Suppose we start from the state $T = \{1, 2, 3\}$. The inhibitor sets belonging to the reactions immediately block any application of a reaction, hence, $res_{\mathcal{A}}(T) = \emptyset$.

On the other hand, starting from the state $\{2, 3\}$, we have that $res_{\mathcal{A}}(T) = \{1\}$. Continuing in this way, we obtain the following sequence of states: $\{2, 3\} \to \{1\} \to \{2\} \to \{1, 3\} \to \{2, 3\} \to \ldots$, which yields an infinite evolution, as the sequence of states eventually becomes periodic.

Since there are only finitely many states, these two cases exhaust all possible behaviours that reaction systems can exhibit.

In Pawlak's original definition (cf. [15]), upper and lower approximations are induced by an equivalence relation on the underlying set. The equivalence classes form the base sets, whose unions constitute the definable sets. The lower approximation of a given set is defined as the largest definable set contained in it, while its upper approximation is the smallest definable set containing it. In other terms, the lower approximation comprises the union of all contained equivalence classes, whereas the upper approximation can be described as the union of all equivalence classes having a nonempty intersection with the given set.

**Definition 2 (Lower and upper approximation).**

1. *Let $S$ be a nonempty set and $R$ be an equivalence relation on $S$. Let us denote $[a] = \{b \in S \mid aRb\}$. The collection of equivalence classes with respect to $R$ is called the set of base sets, which is, in other words, the partition set for $R$. We write $[R]$ for this set. Let $A \subseteq S$. The lower approximation and upper approximation of $A \subseteq S$ are, respectively, defined as*
$$\underline{A}_R = \{B \in [R] \mid B \subseteq A\}$$
*and*
$$\overline{A}_R = \{B \in [R] \mid B \cap A \neq \emptyset\}.$$
*The subscript is omitted when $R$ is clear from the context.*
2. *The set of finite unions of equivalence classes is called the collection of definable sets. We denote this set by $D_R$. When a set is not necessarily definable, we call it non-definable.*
3. *Let $a = (R_a, I_a, P_a) \in A$. If both of $R_a, I_a$ are definable, then we call a core-definable. Moreover, if the product of $a$ is definable, then we call a product-definable, and, if each of $R_a, I_a, P_a$ are definable, then we call a definable. The set of reactions $A$ is (core, product) definable provided every $a \in A$ is (core, product) definable.*

The next definition establishes the connection between reaction systems and approximations.

**Definition 3 (Definite and possible result of a reaction).**

1. *Let $a = (R_a, I_a, P_a) \in A$. Let $X = (S, R)$ be an underlying rough set. Assume $T \subseteq S$. We say that $a$ is definitely enabled– in notation: $den_a(T)$– if $\overline{R}_a \subseteq \underline{T}_R$ and $\overline{I}_a \cap \overline{T}_R = \emptyset$. In this case, we define the definite result of $a$ as before. Namely,*
$$dres_a(T) = \begin{cases} P_a & \text{if } a \text{ is definitely enabled,} \\ \emptyset & \text{otherwise.} \end{cases}$$
2. *We define the possible result of $\mathcal{A}$ as follows. Let $a = (R_a, I_a, P_a) \in A$. Assume $T \subseteq S$. We say that $a$ is possibly enabled– in notation: $pen_a(T)$– if $\underline{R}_a \subseteq \overline{T}_R$ and $\underline{I}_a \cap \underline{T}_R = \emptyset$. In this case, we define the definite result of $a$ as before. Namely,*
$$pres_a(T) = \begin{cases} P_a & \text{if } a \text{ is possibly enabled,} \\ \emptyset & \text{otherwise.} \end{cases}$$

Let $\mathcal{A} = (S, A)$ be a reaction system. If $R \subseteq S \times S$ is the underlying equivalence relation, then the reaction system $\mathcal{A}$ together with the induced approximation space will be denoted by $\mathcal{A}_R = (S, A, R)$. The structure $\mathcal{A}_R = (S, A, R)$ will be called a reaction system with rough sets, or a rough reaction system, where $\mathcal{A} = (S, A)$ is the standard or regular "part" of the reaction system $\mathcal{A}_R$ with rough sets. We note that any reaction system can be regarded as a reaction system with rough sets by taking the identity relation $=$ as the equivalence relation on $S$.

**Definition 4.** *[Definite and possible result of a reaction set] Assume $\mathcal{A}_R = (S, A, R)$ is a reaction system with rough sets.*

1. *We define the definite result of $\mathcal{A}_R$ on $T$ as follows:*

$$dres_{\mathcal{A}_R}(T) = \bigcup_{\{a \in A | den_a(T)\}} dres_a(T).$$

2. *We define the possible result of $\mathcal{A}_R$ on $T$ in the following way:*

$$pres_{\mathcal{A}_R}(T) = \bigcup_{\{a \in A | pen_a(T)\}} pres_a(T).$$

*Given a rough reaction system $\mathcal{A}_R = (S, A, R)$, as before, $res_{\mathcal{A}_R}(T)$ will be referred to as the result of the application of $\mathcal{A}$ to $T$, where $res_{\mathcal{A}_R}(T)$ is computed according to Definition 1. The subscript $R$ may be omitted if it is clear from the context.*

## 3 Variants of rough reaction systems

It may happen that, given a transition system, the evolution of the transition system cannot be fully controlled or accurately observed. Instead, we are often limited to the incomplete picture provided by our perceptions.

We provide an example below to illustrate this phenomenon. The example concerns chemical substances that may react with one another. However, because our observations are imperfect, we are unable to distinguish certain substances from each other. As a result, the exact outcome of a sequence of reactions cannot be determined, and we must instead settle for an approximate characterization of its possible outcomes.

*Example 2.* Let $S = \{a, A, B, c, C\}$, where the letters denote different chemical substances. We know that substances $a, A$ and $c, C$ are indistinguishable. This is expressed by the definition of the equivalence relation $R$, which we present by the equivalence classes: $[R] = \{\{a, A\}, \{B\}, \{c, C\}\}$. Let $\mathcal{A} = (S, A)$, where $A = \{a_1 : (\{A\}, \{c\}, \{A, c\}), a_2 : (\{a\}, \{C\}, \{a, A, C\})\}$. Let $T \subseteq S$ be $\{a\}$. Then

$$\{a\} \to res_{\{a_2\}} = \{a, A, C\} \to res_{\{a_1\}} = \{A, c\},$$

where the subscripts indicate the reaction set that can actually be applied. On the other hand, if $a$ and $A$, and $c$ and $C$, respectively, are indistinguishable to us, then we are not sure whether $a_1$ or $a_2$ is enabled from the start and which one of $a_1$ and $a_2$ is enabled in the next step. In particular, we are unable to determine whether the system eventually reaches a halting state.

Starting from $T = \{a\}$, we can conclude that $\underline{T} = \emptyset$, while $\overline{T} = \{a, A\}$. This means that neither of $a_1$ and $a_2$ are definitely enabled, while both $a_1$ and $a_2$ are possibly enabled. Hence, $dres_{\mathcal{A}}(T) = \emptyset$ and $pres_{\mathcal{A}}(T) = \{a, A, C\}$ since, in the latter case, neither of them are blocked by their respective inhibitors.

There are several possible situations that can be imagined in connection with in what respect can our assessment be imperfect. We will identify some of them below. The first case is the most general one, where both the rule set and the underlying set are composed entirely of non-definable constituents. The subsequent two cases correspond to situations in which some of the constituents are precisely known, being definable sets with respect to the underlying partition. Let $\mathcal{A}_R = (S, A, R)$ be a reaction system together with an equivalence relation. All the cases are concerned with an imaginary situation when either the underlying set or the constituents of the reactions are definable or non-definable. Based on our limited perception, we form expectations about the possible outcomes of the reaction system. In particular, we may seek to characterize either the certain or the possible results produced by the system by imposing different conditions on the prediction of rule enabledness.

(A) Neither $A$, nor $S$ are definable: Imagine that we are given an experimental setting together with a collection of reaction rules. However, our perception of the system is limited, and objects can only be distinguished up to the granularity induced by the underlying approximating sets. The same applies to the reaction system itself: although it evolves according to its prescribed rules, the reactants and inhibitors are available to us only through their approximations. More precisely, we know only their upper and lower approximations, from which we infer the reactions whose combined effects constitute the outcome of a reaction step. In this setting, our goal is to formulate upper and lower estimates of the evolution of the reaction system based solely on the approximate information available about the underlying set.

(B) $A$ is core-definable and $S$ is non-definable: We are given a set of reactions that are exact by description: the sets $R_a$ and $I_a$ governing the applicability of each rule $a \in A$ are definable, hence, we know their exact values, which are definable sets. However, the object set, and the product set of the reactions, respectively, can be perceived only up to the level of detail provided by the approximations.

(C) $A$ is product-definable and $S$ is definable: Imagine that we are given a set of objects that are exact by description. However, the applicability of the set of rules manipulating them can be perceived only up to the level of detail provided by the approximation. Again, we have to set upper or lower

estimations regarding the evolution of the reaction system based on our perception of the underlying set.

Several additional cases could also be considered. However, the scenarios discussed above may be regarded as the principal prototypes of how partial information can arise in the computation of reaction systems. Case (A) is the most general setting. Case (B) covers the situation in which the applicability conditions of the reactions are definable, whereas Case (C) corresponds to the situation in which the products of the reactions are definable.

## 4 Results

This section presents results on reaction systems equipped with approximation spaces. First, we consider the most general case, where neither the reaction system nor the underlying set are definable. Our first concern is the extent to which an approximation by rough sets can predict the outcome of a reaction system after several computational steps. This part is similar to our previous work [4], although the underlying approximation space is now a partition induced by an equivalence relation. This gives rise to substantial differences in the treatment of Cases (B) and (C).

The following proposition holds in all the cases under considered in the previous section.

**Proposition 1.** *Let $\mathcal{A}_R = (S, A, R)$ be a reaction system with approximation space. Let $T \subseteq S$. Then*

$$dres_{\mathcal{A}_R}(T) \subseteq res_{\mathcal{A}_R}(T) \subseteq pres_{\mathcal{A}_R}(T).$$

*Proof.* Straightforward.

The relations described above can, however, be quite fragile. We will show below that there exists a rough reaction system for which the outcomes produced by the reaction system differ significantly depending on whether they are computed when we regard the reaction system as a regular one or the result is obtained based on our approximations relying on the given relation $R$. This holds even if we consider only two successive steps of reaction system applications. To this end, we first introduce the necessary definitions, which are borrowed from [4]. Since we are concerned with the behaviour of reaction systems over several successive computational steps, we extend the notion of a function computed by a reaction system introduced in [17] to the case of $k$-fold application. In what follows, if $S$ is any given set, then let us denote by $\mathcal{S}_1 = 2^S \setminus \{\emptyset\}$.

**Definition 5.** *[Functions $k$-computed by reaction systems] Let $\mathcal{A} = (S, A)$ be a reaction system over the finite set $S$. We determine when $\mathcal{A}$ $k$-computes the (partial) function $f : \mathcal{S}_1 \to \mathcal{S}_1$. In case of $k = 1$, we simply say that $\mathcal{A}$ computes $f$. Let $T \in \mathcal{S}_1$.*

- $k = 0$*: In this case,* $f(T) = T$.
- $k = 1$*: Let* $T$ *be such that there exists* $a \in A$ *for which* $en_a(T)$ *holds. Then we call* $res_{\mathcal{A}}$ *meaningful on* $T$ *and we write* $f(T) = res_{\mathcal{A}}(T)$*. Otherwise, let* $f(T)$ *be undefined. In the case of* $k = 1$*, we say that* $\mathcal{A}$ *computes the (partial) function* $f$.
- $k > 1$*: Assume there exists* $a \in A$ *for which* $en_a(res_{\mathcal{A}}^{k-1}(T))$ *holds. Then* $res_{\mathcal{A}}^k$ *is meaningful on* $T$ *and we define* $f(T) = res_{\mathcal{A}}^k(T) = res_{\mathcal{A}}(res_{\mathcal{A}}^{k-1}(T))$*. Otherwise, let* $f(T)$ *be undefined. In this case we say that* $\mathcal{A}$ $k$*-computes the (partial) function* $f$.

*In notation:* $f = res_{\mathcal{A}}^k$ *or* $f = res^k$ *if* $\mathcal{A}$ *is clear from the context.*

**Definition 6.** *[Functions computed with external elements] Let* $f : \mathcal{S}_1 \to \mathcal{S}_1$ *be a partial function. Let* $C$ *be a finite set. A reaction system* $\mathcal{A}' = (S', A)$*, where* $S' = S \cup C$*, computes* $f$ *with external elements from* $C$ *and input elements* $D \subseteq C$ *if, for every* $T \subseteq S$*,* $f' : \mathcal{S}_1' \to \mathcal{S}_1'$ *is computed by* $\mathcal{A}'$ *on* $T \cup D$ *and* $f(T) = f'(T \cup D) \cap S$ *in the sense that either both functions are undefined or both are defined and the values on the left and right hand side of the equation are equal. The notion of* $k$*-computation with external elements can be defined along the same lines as in Definition 5.*

We note that we impose no restrictions on the set $C$. For convenience, we may assume that $S$ and $C$ are disjoint. Observe that computing with external elements seems to constitute a much weaker form of interference with the computation of a reaction system than the notion of interactive processes [11], since, in the former case, only the initial state involves additional elements, while the remainder of the computation is determined solely by the initial state. We extend the notions just defined to the case of definite and possible results.

**Definition 7.** *[Functions computed with rough reaction systems] The above notions apply mutatis mutandis when we consider a rough reaction system. In this case, we can define the definite (possible, resp.) result using the operators* $dres_{\mathcal{A}_R}$ *(*$pres_{\mathcal{A}_R}$*, resp.). We denote the* $i$*-fold application (*$i \geq 1$*) of* $dres_{\mathcal{A}_R}$ *(*$pres_{\mathcal{A}_R}$*, resp.) on* $T$ *by* $dres_{\mathcal{A}_R}^i(T)$ *(*$pres_{\mathcal{A}_R}^i(T)$*, resp.). We say that the function obtained in this way is definitely (resp. possibly)* $i$*-computed by* $\mathcal{A}_R$*. The notions also extend to computation with external elements.*

The next theorem shows that, in rough reaction systems, the predicted outcome may differ significantly from the actual outcome. We are concerned with the connection of the definite and standard results first.

**Theorem 1.** *Let* $\mathcal{A} = (S, A)$ *be a reaction system over the finite set* $S$*. Assume that* $\mathcal{A}$ *computes the (partial) function* $f : \mathcal{S}_1 \to \mathcal{S}_1$*. Let* $s \in S$ *be a fixed element of* $S$*. Let* $g_s : \mathcal{S}_1 \to \mathcal{S}_1$ *be the (partial) function such that* $f$ *is defined on any* $T \in \mathcal{S}_1$ *iff* $g_s$ *is defined on* $T$ *and, in this case,* $g_s(T) = \{s\}$*. Then there exists* $S \subseteq S'$ *and an equivalence relation* $R$ *such that, for* $\mathcal{A}_R' = (S', A', R)$*, the following properties hold:*

1. *The function $f$ is 2-computed (with external elements) by $\mathcal{A}'_R$.*
2. *The function $g_s$ is definitely 2-computed by $\mathcal{A}'_R$.*

*Proof.* Let $S$ and $\mathcal{A} = (S, A)$ be as above. Let $f : \mathcal{S}_1 \to \mathcal{S}_1$ be the (partial) function computed by $\mathcal{A}$. We construct $\mathcal{A}'_R$ in the following way. Let $S' = S \cup \{u, v\} \cup \{e_a \mid a \in A\}$. The reactions are defined as follows:

$$\begin{aligned} A' = &\{a' : (R_a, I_a, \{e_a\}) \mid a : (R_a, I_a, P_a) \in A\} \ \cup \\ &\{p_a : (\{e_a, u\}, \emptyset, P_a) \mid a \in A\} \ \cup \\ &\{q_a : (\{e_a\}, \{u\}, \{s\}) \mid a \in A\} \ \cup \\ &\{r_v : (\{v\}, \{u\}, \{u\})\}. \end{aligned}$$

Now we provide the equivalence relation $R$, which we describe below by defining the partitions. We let $u$ be similar to $v$ and otherwise let the approximation space be the discrete one. Namely,

$$\begin{aligned} [R] = &\{\{p\} \mid p \in S\} \cup \\ &\{\{u, v\}\} \cup \{\{e_a\} \mid a \in A\}. \end{aligned}$$

The behaviour of $\mathcal{A}'_R$ as a regular reaction system can be described in the following way. In what follows, we omit the subscript $R$. Let $f'$ be the function 2-computed by $\mathcal{A}'$. Assume $T \subseteq S$ such that $f(T)$ is defined. Then we start $\mathcal{A}'$ with $T' = T \cup \{v\}$. If $\mathcal{A}'$ is considered a regular reaction system, then all the $a'$s are executed for which $en_{a'}(T')$. By definition, $en_{a'}(T')$ iff $en_a(T)$. Since $f(T)$ is defined, $en_{a'}(T')$ will hold for at least one $a' \in A'$, hence $res_{\mathcal{A}'}(T')$ will be nonempty. Furthermore, $(\{v\}, \{u\}, \{u\})$ is applicable, which means that $res_{\mathcal{A}'}(T') = \bigcup_{en_{a'}(T')} P_{a'} \cup \{u\} = \{e_a \mid en_a(T)\} \cup \{u\}$. Thus $res^2_{\mathcal{A}'}(T') = \bigcup_{en_a(T)} P_a$, since the only rules applicable in the second step are the rules $p_a$ provided $en_a(T)$ holds. This yields $f'(T) = res^2_{\mathcal{A}'}(T') = res_{\mathcal{A}}(T) = f(T)$. If $f(T)$ is undefined, then, since for no $a \in A$ $en_a(T)$ holds, $en_{a'}(T')$ is undefined for any $a' \in A'$. In this case, $res_{\mathcal{A}'}(T') = \{u\}$ for any $T \subseteq S$, hence $res^2_{\mathcal{A}'} = \emptyset$, which implies that $f'(T)$ is undefined.

Now, we compute the definite result. Let, as before, $T \subseteq S$ be given. Assume $f$ is defined on $T$. Let $T' = T \cup \{v\}$ and let $f''$ denote the function definitely 2-computed by $\mathcal{A}'$. We have $\overline{R}_a = R_a$ and $\overline{I}_a = I_a$ for every $a \in A$. This implies that $en_{a'}(T')$ holds for any $a' \in A'$ iff $en_a(T)$ is true. Hence, $dres_{\mathcal{A}'}(T')$ will contain all the $e_a$ for which $en_a(T)$ holds. On the other hand, $\overline{\{v\}} = \{u, v\}$, hence, $\overline{T \cup \{v\}} = T \cup \{u, v\}$ for any $T \subseteq S$. Hence, $r_v : (\{v\}, \{u\}, \{u\})$ is not applicable in this case due to its inhibitor set. This implies that the rules $\{q_a\}$ $(a \in A)$, whenever $e_a \in dres_{\mathcal{A}'}(T')$, are applicable in the second stage and the rules $p_a$ $(a \in A)$ are not applicable. We can conclude that $dres^2_{\mathcal{A}'}(T') = \{s\}$. On the other hand, if $f$ is not defined on $T$, then $dres^2_{\mathcal{A}'}(T') = \emptyset$, and, in this case, $f''$ is undefined.

If we consider the possible result of a reaction system, we arrive at an analogous conclusion. We state the result below together with its proof, which proceeds along the same lines as the previous one.

**Theorem 2.** *Let $\mathcal{A} = (S, A)$ be a reaction system over the finite set $S$. Assume that $\mathcal{A}$ computes the (partial) function $f : \mathcal{S}_1 \to \mathcal{S}_1$. Let $s \in S$ be a fixed element of $S$. Let $g_s : \mathcal{S}_1 \to \mathcal{S}_1$ be the (partial) function such that $f$ is defined on any $T \in \mathcal{S}_1$ iff $g_s$ is defined on $T$ and, in this case, $g_s(T) = \{s\}$. Then there exists $S \subseteq S'$ and an equivalence $R$ such that, for $\mathcal{A}'_R = (S', A', R)$, the following properties hold:*

1. *The function $f$ is 2-computed (with external elements) by $\mathcal{A}'_R$.*
2. *The function $g_s$ is possibly 2-computed by $\mathcal{A}'_R$.*

*Proof.* Let $S$ and $\mathcal{A} = (S, A)$ be as above. Let $f : \mathcal{S}_1 \to \mathcal{S}_1$ be the(partial) function computed by $\mathcal{A}$. We construct $\mathcal{A}'_R$ in the following way. Let $S' = S \cup \{u, v, w\} \cup \{e_a \mid a \in A\}$. In the definition below, $T$ ranges over the subsets of $S$. Then the reactions are

$$\begin{aligned} A' = {} & \{a' : (R_a, I_a, \{e_a\}) \mid a : (R_a, I_a, P_a) \in A\} \cup \\ & \{r_{u,v} : (\{u, v\}, \{w\}, \{w\})\} \cup \\ & \{p_a : (\{e_a\}, \{w\}, P_a) \mid a \in A\} \cup \\ & \{(q_a : \{w\}, \{s\}, \{s\}) \mid a \in A\}. \end{aligned}$$

Let us consider the following partition set $[\mathcal{R}]$ stemming from the underlying relation $R$.

$$\begin{aligned} [R] = {} & \{\{p\} \mid p \in S\} \cup \\ & \{\{u, v\}, \{w\}\} \cup \{\{e_a\} \mid a \in A\}. \end{aligned}$$

We claim that $\mathcal{A}'_R$ as a reaction system 2-computes $f$, whereas $\mathcal{A}'_R$ based on the equivalence $R$ possibly 2-computes $g_s$. We omit the subscript $R$ below.

The behaviour of $\mathcal{A}'$ as a reaction system can be illustrated as follows. Assume $T \subseteq S$ such that $f(T)$ is defined. Let $f'$ be the function 2-computed by $\mathcal{A}'$. Then we start $\mathcal{A}'$ with $T' = T \cup \{v\}$. In this case, all the $a'$s are executed for which $en_{a'}(T')$. By definition, $en_{a'}(T')$ iff $en_a(T)$. Since $f(T)$ is defined, $en_{a'}(T')$ will hold for at least one $a' \in A'$. Furthermore, $(\{u, v\}, \emptyset, \{w\})$ is repressed, which means that $res_{\mathcal{A}'}(T') = \bigcup_{en_{a'}(T')} P_{a'} = \{e_a \mid en_a(T)\}$. Then, by the rules $p_a$ $(a \in A)$, $f'(T) = res^2_{\mathcal{A}'}(T') = \bigcup_{en_a(T)} P_a = f(T)$. If $f(T)$ is undefined, then, since, for no $a \in A$, $en_a(T)$ holds, $en_{a'}(T')$ is undefined for any $a' \in A'$. In this case, $res_{\mathcal{A}'}(T') = \emptyset$, hence $f'(T')$ is undefined.

On the other hand, let us consider the case when $pres^2_{\mathcal{A}'}$ is evaluated using the partition set $[R]$. Let $T \subseteq S$. Assume $f$ is defined on $T$. Let $T' = T \cup \{v\}$. We have $\underline{R}_a = R_a$ and $\underline{I}_a = I_a$ for every $a \in A$. This implies that $en_{a'}(T')$

holds for any $a' \in A'$ iff $en_a(T)$ is true. Hence, $pres_{\mathcal{A}'}(T')$ will contain all the $e_a$ for which $en_a(T)$ is true. On the other hand, $\{u, v\} \subseteq \overline{T'} = T \cup \{u, v\}$ for any $T \subseteq S$, hence $(\{u, v\}, \{w\}, \{w\})$ is applicable in this case. This involves that the rules $\{q_a \mid a \in A\}$, whenever $e_a \in pres_{\mathcal{A}'}(T')$, are applicable in the second step. Furthermore, the rules $p_a$ $(a \in A)$ are forbidden because of the presence of $w$. Thus, $pres^2_{\mathcal{A}'}(T') = \{s\}$. If $f$ is not defined on $T$, then $pres^2_{\mathcal{A}'}(T') = \emptyset$, hence, the function possibly 2-computed by $\mathcal{A}'$ is undefined on $T'$.

We remark that, in addition, all the rules are core-definable in the proof above with respect to the given partition set.

In what follows, we investigate how the exactness of approximations changes as the definability assumptions on the basic constituents are modified. In particular, we examine situations analogous to those considered in the above theorems for Cases (B) and (C). It turns out that, in the presence of an equivalence relation, the approximations can be unexpectedly exact in certain situations. We explore these phenomena below.

We begin with Case (B), namely, when the given rough reaction system is core-definable. The notions of definite and possible executions are defined exactly as in Definition 4. We first establish the somewhat surprising result that, in this case, the definite execution exactly captures the evolution of the reaction system. More precisely, the following theorem holds.

**Theorem 3.** *Let $\mathcal{A}_R = (S, A, R)$ be a core-definable rough reaction system, and let $T \subseteq S$. Then the definite result of $\mathcal{A}_R$ on $T$ coincides with the actual outcome of $\mathcal{A}_R$ on $T$.*

*Proof.* Let $\mathcal{A}_R = (S, A, R)$ be a core-definable rough reaction system, and let $T \subseteq S$. By Definition 4, the definite result of $\mathcal{A}_R$ on $T$ is obtained by taking the union of the product sets $P_a$ over all (definitely) applicable reactions $a$. A reaction $a = (R_a, I_a, P_a)$ is definitely applicable, in this particular case, if $R_a \subseteq \underline{T}$ and $I_a \cap \overline{T} = \emptyset$. Since $\mathcal{A}_R$ is core-definable, the reactant and inhibitor sets $R_a$ and $I_a$ are definable.

We show that applicability in this sense coincides with applicability when $\mathcal{A}_R$ is regarded as a regular reaction system. More precisely, both conditions are equivalent to the standard applicability conditions in the regular case.

- $R_a \subseteq \underline{T}$. First, $R_a \subseteq \underline{T}$ obviously implies $R_a \subseteq T$. Conversely, assume that $R_a \subseteq T$. Since $R_a \in D_R$, we have $R_a = \underline{R}_a$, and hence $R_a = \underline{R}_a \subseteq \underline{T}$.
- $I_a \cap \overline{T} = \emptyset$. Obviously, this implies that $I_a \cap T = \emptyset$. Conversely, assume that $I_a \cap T = \emptyset$. Let $t \in I_a \cap \overline{T}$. Then there exists $s \in T$ such that $[t] = [s]$. Since $I_a \in D_R$, we have $[s] \subseteq I_a$, which implies $s \in I_a$, and thus $s \in I_a \cap T$, a contradiction.

This shows that the enabling conditions are equivalent in both cases. Consequently, the definite approximation coincides exactly with the actual result of the reaction system.

Regarding the possible result, the remark at the end of the proof of Theorem 2 shows that, in this case, the approximation may deviate significantly from the actual state.

Now we turn to Case (C). Namely, we assume that $\mathcal{A}$ is product-definable, and the underlying set $S$ is definable. We show that, in this case, the definite execution exactly captures again the evolution of the reaction system. More precisely, the following theorem holds.

**Theorem 4.** *Let $\mathcal{A}_R = (S, A, R)$ be a product-definable rough reaction system, and let $T \subseteq S$ be definable. Then the definite result of $\mathcal{A}_R$ on $T$ coincides with the actual outcome of $\mathcal{A}_R$ on $T$.*

*Proof.* Let $\mathcal{A}_R = (S, A, R)$ be a product-definable rough reaction system, and let $T \subseteq S$. We assume that both $S$ and $T$ are definable. By Definition 4, the definite result of $\mathcal{A}_R$ on $T$ is obtained by taking the union of the product sets $P_a$ over all definitely applicable reactions $a$. A reaction $a = (R_a, I_a, P_a)$ is definitely applicable in this particular case if $\underline{R}_a \subseteq T$ and $\underline{I}_a \cap T = \emptyset$. Since $\mathcal{A}_R$ is product-definable, the product set $P_a$ together with the underlying set $S$ and the subset $T$ on which $a$ is applied are definable.

We show that applicability, in this sense, coincides with applicability when $\mathcal{A}_R$ is regarded as a regular reaction system. More precisely, both conditions are equivalent to the standard applicability conditions in the regular case.

- $\overline{R}_a \subseteq T$. First, $\overline{R}_a \subseteq T$ obviously implies $R_a \subseteq T$. Conversely, assume that $R_a \subseteq T$. Then $R_a \subseteq \overline{R}_a \subseteq \overline{T} = T$.
- $\overline{I}_a \cap T = \emptyset$. Obviously, this implies that $I_a \cap T = \emptyset$. Conversely, assume that $I_a \cap T = \emptyset$. Let $t \in \overline{I}_a \cap T$. Then there exists $p \in I_a$ such that $[t] = [p]$. Since $T \in D_R$, we have $[p] \subseteq T$, which implies $p \in I_a \cap T$, a contradiction. Hence, $\overline{I}_a \cap T = \emptyset$ if and only if $I_a \cap T = \emptyset$.

This shows that the enabling conditions are equivalent in both cases. Consequently, the definite approximation in the product-definable case coincides exactly with the actual outcome of the reaction system.

Now we examine the possible result in Case (C). We have the following theorem.

**Theorem 5.** *Let $\mathcal{A} = (S, A)$ be a reaction system over the finite set $S$. Assume that $\mathcal{A}$ computes the (partial) function $f : \mathcal{S}_1 \to \mathcal{S}_1$. Let $s \in S$ be a fixed element of $S$. Let $g_s : \mathcal{S}_1 \to \mathcal{S}_1$ be the (partial) function such that $f$ is defined on any $T \in \mathcal{S}_1$ iff $g_s$ is defined on $T$ and, in this case, $g_s(T) = \{s\}$. Then there exists $S \subseteq S'$ and an equivalence $R$ such that, for $\mathcal{A}'_R = (S', A', R)$, the following properties hold:*

1. *$\mathcal{A}'_R$ is product-definable,*
2. *the function $f$ is 2-computed (with external elements) by $\mathcal{A}'_R$,*
3. *the function $g_s$ is possibly 2-computed by $\mathcal{A}'_R$.*

*Proof.* The proof is very similar to that of Theorem 2. We provide the details. Let $S$ and $\mathcal{A} = (S, A)$ be as above. Let $f : \mathcal{S}_1 \to \mathcal{S}_1$ be the (partial) function computed by $\mathcal{A}$. We construct $\mathcal{A}'$ in the following way. Let $S' = S \cup \{u, v, w\} \cup \{e_a \mid a \in A\}$. Then we set

$$\begin{aligned} A' = {} & \{a' : (R_a, I_a, \{e_a\}) \mid a : (R_a, I_a, P_a) \in A\} \cup \\ & \{r_u : (\{u\}, \{w\}, \{w\})\} \cup \\ & \{p_a : (\{e_a\}, \{w\}, P_a) \mid a \in A\} \cup \\ & \{(q_a : \{e_a, w\}, \emptyset, \{s\}) \mid a \in A\}. \end{aligned}$$

Let us consider the following partition set $[\mathcal{R}]$ stemming from the underlying relation $R$.

$$\begin{aligned} [R] = {} & \{\{p\} \mid p \in S\} \cup \\ & \{\{u, v\}, \{w\}\} \cup \{\{e_a\} \mid a \in A\}. \end{aligned}$$

We claim that $\mathcal{A}' = (S', A', R)$ 2-computes $f$ as a standard reaction system, whereas, when relying on the approximation induced by the equivalence relation $R$, $\mathcal{A}'$ may 2-compute $g_s$.

The computation of $\mathcal{A}'$ as a reaction system can be illustrated as follows. Assume $T \subseteq S$ such that $f(T)$ is defined. Then we start $\mathcal{A}'$ with $T' = T$. Let $f'$ be the partial function 2-computed by $\mathcal{A}'$. Observe that, since $en_{a'}(T')$ iff $en_a(T)$, all the $a'$s are executed for which $en_a(T)$. Since $f(T)$ is defined, $en_a(T)$ will hold for at least one $a \in A$. Hence, at least one $a' \in A'$ is enabled. Furthermore, $(\{u\}, \{w\}, \{w\})$ is not enabled, which means that $res_{\mathcal{A}'}(T') = \bigcup_{en_{a'}(T')} P_{a'} = \{e_a \mid en_a(T)\}$. Then, by the rules $p_a$ ($a \in A$), $f'(T') = res^2_{\mathcal{A}'}(T') = \bigcup_{en_a(T)} P_a = f(T)$ in the second step. If $f(T)$ is undefined, then, since, for no $a \in A$, $en_a(T)$ holds, $en_{a'}(T')$ is undefined for every $a' \in A'$. This implies that no reaction is enabled, and hence $res_{\mathcal{A}'} = \emptyset$, which means that $f'(T')$ is undefined.

On the other hand, let us consider the case when $pres^2_{\mathcal{A}'}$ is evaluated using the partition set $[R]$. First, observe that $\mathcal{A}'$ is product-definable. Let $T \subseteq S$. Again, set $T' = T$ and let $f''$ be the function possibly 2-computed by $\mathcal{A}'$. Assume $f$ is defined on $T$. We have $R_{a'} = R_a$ and $I_{a'} = I_a$ for every $a \in A$. This implies that $en_{a'}(T')$ holds for any $a \in A$ iff $en_a(T)$ is true. Hence, $pres_{\mathcal{A}'}(T')$ will contain all the $e_a$ for which $en_a(T)$ is valid. On the other hand, $\underline{\{u\}} = \emptyset \subseteq T' = T$ for any $T \subseteq S$. This means that $r_u$ is activated and $w$ will be an element of $pres_{\mathcal{A}'}(T')$. This involves that the rules $\{q_a \mid a \in A\}$, whenever $e_a \in pres_{\mathcal{A}'}(T')$, are applicable in the second step. Furthermore, the rules $p_a$ ($a \in A$) are forbidden because of the presence of $w$. Thus, $pres^2_{\mathcal{A}'}(T') = \{s\}$. If $f$ is not defined on $T$, then $pres_{\mathcal{A}'}(T') = \{w\}$, and hence, $pres^2_{\mathcal{A}'}(T') = \emptyset$. This involves that $f''(T')$ is undefined.

The above discussion shows that, in Cases (B) and (C), computing the definite approximations yields the exact evolution of the underlying reaction system, whereas the possible approximation may still lead to significant deviations.

This naturally raises the question of whether there exist conditions on the underlying equivalence relation or on the reaction system that guarantee the approximations behave as expected, namely, $dres^k_{\mathcal{A}}(T) \subseteq res^k_{\mathcal{A}}(T) \subseteq pres^k_{\mathcal{A}}(T)$ for every $k \in \mathbb{N}$, where $\mathcal{A} = (S, A, R)$ is a given rough reaction system and $T \subseteq S$ is arbitrary. Leaving this question for future work, we provide a simple case in which the above inclusions hold.

**Definition 8.** *Let $S$ be a finite set. Let $\mathcal{A} = (S, A)$ be a reaction system on $S$. We say that $\mathcal{A}$ is monotone if, for all $T_1, T_2 \subseteq S$, $T_1 \subseteq T_2$ implies $res_{\mathcal{A}}(T_1) \subseteq res_{\mathcal{A}}(T_2)$ whenever both $res_{\mathcal{A}}(T_1)$ and $res_{\mathcal{A}}(T_2)$ are defined.*

We have the following simple statement.

**Theorem 6.** *Let $S$ be a finite set. Let $\mathcal{A}_R = (S, A, R)$ be a rough reaction system on $S$. Assume $\mathcal{A}_R$, as a regular reaction system, is monotone. Let $T \subseteq S$. Then*

$$dres^k_{\mathcal{A}_R}(T) \subseteq res^k_{\mathcal{A}_R}(T) \subseteq pres^k_{\mathcal{A}_R}(T) \qquad (*)$$

*holds for every $k \in \mathbb{N}$.*

*Proof.*

- $k = 1$. The assertion is true by definition.
- $k > 1$. Assume that $dres^{k-1}_{\mathcal{A}_R}(T) \subseteq res^{k-1}_{\mathcal{A}_R}(T) \subseteq pres^{k-1}_{\mathcal{A}_R}(T)$ holds. Let $a \in A$ be arbitrary. Then Proposition 1, together with the monotonicity assumption, yield the result.

A comprehensive treatment of this question will likely necessitate the study of the relations of functional equivalence or functional cover between different reaction sets, similarly to the approach taken in the paper [14]. We leave the determination of more refined conditions under which the relations $(*)$ hold for future work.

## 5 Conclusion

In this paper, we studied reaction systems equipped with a partition set induced by an equivalence relation. Based on the standard lower and upper approximations determined by this relation, we investigated how closely the predicted behaviour can approximate the actual evolution of a reaction system. We explored several scenarios depending on the extent to which the reaction system or the underlying finite set is definable. We showed that, in the general case, even after as few as two evolutionary steps, the state of the original reaction system may become indeterminable from its approximants. However, in more specific cases, the lower approximation provides an exact prediction of the reaction system. Finally, we posed as an open question the problem of determining under which circumstances the approximations provide useful information about the evolution of the reaction system even after several computational steps.

## References


1. Abraham, A., Falcón, R., Bello, R.: Rough Set Theory: A True Landmark in Data Analysis. Springer, Berlin, Heidelberg (2009). https://doi.org/https://doi.org/10.1007/978-3-540-89921-1
2. Azimi, S., Gratie, C., Ivanov, S., Manzoni, L., Petre, I., Porreca, A.E.: Complexity of model checking for reaction systems. Theoretical Computer Science **623**, 103–113 (2016). https://doi.org/https://doi.org/10.1016/j.tcs.2015.11.040, https://www.sciencedirect.com/science/article/pii/S0304397515011263
3. Banerjee, M., Chakraborty, M.K., Szałas, A.: Logics from rough sets. Journal of Applied Non-Classical Logics **34**(2-3), 171–173 (2024). https://doi.org/10.1080/11663081.2024.2336616
4. Battyányi, P.: Reaction systems with approximation spaces. In: Machines, Computations, and Universality 11th International Conference, MCU 2026, Trier, Germany, July 27–29, 2026. Springer International Publishing, to appear.
5. Brodo, L., Bruni, R., Falaschi, M.: A logical and graphical framework for reaction systems. Theoretical Computer Science **875**, 1–27 (2021). https://doi.org/https://doi.org/10.1016/j.tcs.2021.03.024, https://www.sciencedirect.com/science/article/pii/S0304397521001778
6. Campagner, A., Ciucci, D., Dorigatti, V.: Approximate reaction systems based on rough set theory. In: Bello, R., Miao, D., Falcón, R., Nakata, M., Rosete, A., Ciucci, D. (eds.) Rough Sets. pp. 48–60. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-52705-1_4
7. Corolli, L., Maj, C., Marini, F., Besozzi, D., Mauri, G.: An excursion in reaction systems: From computer science to biology. Theoretical Computer Science **454**, 95–108 (2012). https://doi.org/https://doi.org/10.1016/j.tcs.2012.04.003, formal and Natural Computing
8. Dennunzio, A., Formenti, E., Manzoni, L., Porreca, A.E.: Complexity of the dynamics of reaction systems. Information and Computation **267**, 96–109 (2019). https://doi.org/https://doi.org/10.1016/j.ic.2019.03.006, https://www.sciencedirect.com/science/article/pii/S0890540119300240
9. Dutta, S., Jankowski, A., Rozenberg, G., Skowron, A.: Linking reaction systems with rough sets. Fundamenta Informaticae **165**, 283 – 302 (2019). https://doi.org/https://doi.org/10.3233/FI-2019-1786, https://api.semanticscholar.org/CorpusID:85562866
10. Düntsch, I.: A logic for rough sets. Theoretical Computer Science **179**(1), 427–436 (1997). https://doi.org/https://doi.org/10.1016/S0304-3975(96)00334-9, https://www.sciencedirect.com/science/article/pii/S0304397596003349
11. Ehrenfeucht, A., Kleijn, J., Koutny, M., Rozenberg, G.: Evolving reaction systems. Theoretical Computer Science **682**, 79–99 (2017). https://doi.org/https://doi.org/10.1016/j.tcs.2016.12.031, https://www.sciencedirect.com/science/article/pii/S030439751730035X, special Issue on Languages and Combinatorics in Theory and Nature
12. Ehrenfeucht, A., Main, M., Rozenberg, G.: Functions defined by reaction systems. International Journal of Foundations of Computer Science **22**(01), 167–178 (2011). https://doi.org/10.1142/S0129054111007927, https://doi.org/10.1142/S0129054111007927
13. Ehrenfeucht, A., Rozenberg, G.: Reaction systems. Fundamenta Informaticae **75**, 263–280 (2007)

14. Genova, D., Hoogeboom, H.J., Kleijn, J.: Functional equivalence and a cover relation for reaction systems. Theoretical Computer Science **1004** (2024). https://doi.org/https://doi.org/10.1016/j.tcs.2024.114633
15. Pawlak, Z.: Rough sets. International Journal of Computer and Information Sciences **11**(5), 341–356 (1982)
16. Polkowski, L., Artiemjew, P.: Rough sets in data analysis: Foundations and applications. In: Smolinski, T.G., Milanova, M.G., Hassanien, A.E. (eds.) Applications of Computational Intelligence in Biology: Current Trends and Open Problems, pp. 33–54. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78534-7_2
17. Salomaa, A.: Functions and sequences generated by reaction systems. Theoretical Computer Science **466**, 87–96 (2012). https://doi.org/https://doi.org/10.1016/j.tcs.2012.07.022
18. Tsumoto, S.: Rough set data analysis. In: Meyers, R.A. (ed.) Computational Complexity: Theory, Techniques, and Applications, pp. 2716–2727. Springer New York, New York, NY (2012). https://doi.org/10.1007/978-1-4614-1800-9_167

# Probabilistic Verification of Robot Controllers using Enzymatic Numerical P Systems Models

Radu Traian Bobe[1][0009−0005−6611−3176], Florentin Ipate[1][0000−0001−8777−3425], and Savas Konur[2][0000−0003−2704−8381]

[1] Department of Computer Science, Faculty of Mathematics and Computer Science University of Bucharest, Str Academiei 14, Bucharest, 010014, Romania
radu.bobe@s.unibuc.ro; florentin.ipate@unibuc.ro
[2] School of Computer Science and Digital Technologies, Aston University, Birmingham, B4 7ET, United Kingdom
s.konur@aston.ac.uk

**Abstract.** The validation of robot controllers constitutes a complex problem: traditional testing and simulation examine only specific execution traces, providing practical evidence without formal guarantees, while exhaustive formal verification becomes computationally intractable due to state space explosion problem. This paper addresses this challenge by presenting a practical approach for verification of obstacle avoidance robot controllers that are based on enzymatic numerical P systems (ENPS). The study demonstrates how model checking can be effectively applied to examine and evaluate the probabilistic behaviour of the controllers. Several properties are formally verified with Monte Carlo Model Checker (MC2) using a logic language called Probabilistic Linear-time Temporal Logic (PLTL), which incorporates temporal operators for describing how the system evolves over time. The results proved that the ENPS model used in our experiments confers the main capabilities to the controllers. Furthermore, we validated that the dynamic generation of weight parameters that are assigned to each proximity sensor of the robot does not compromise the obstacle avoidance behaviour.

**Keywords:** Membrane computing, Numerical P systems, Enzymatic numerical P systems, Robot controllers, Simulation, Model checking, Formal verification.

## 1 Introduction

Over the past two decades, the advancement of *membrane computing* has attracted significant interest and driven innovation in fields like biology, economics, computer science and robotics. This biologically inspired concept defined as a branch of natural computing, has been introduced by Gh. Păun [33] to study parallel, distributed computation inspired by the structure of a living cell. During the last 20 years, membrane computing has achieved both explanatory and applicative purposes, contributing to solving problems from different fields by leveraging biological inspiration. [46] Several variants of membrane systems (P

systems) were proposed [33,35]. The arrangement of the membrane led to different classes of membrane systems. For example, in a *cell-like P system* the membranes are disposed hierarchically. Alternatively, *tissue-like P systems* [45,42] are characterised by the placement of cells at the nodes of a graph, while the environment is modeled as a separate, distinct node. *Spiking neural P systems* represent another category that adopted essential concepts from neurobiology. In a spiking neural P system, *spiking* and *forgetting* rules are applied to a graph of interconnected neurons through synapses [14,20,36,43].
Formal verification allows the checking of system behaviours that cannot be analysed through simulation or testing. It therefore provides additional and often more novel insights into system behaviour. Formal verification has been explored for different categories of P systems. For example, M. Gheorghe et al. [17,18] proposed a method to formally verify the specifications of a P system by first identifying invariants in specification and then use a dedicated tool, called NuSMV to check if the properties as invariants are true or false. The method was applied for some basic categories of P systems with transformation and communication rules. Later on, Ipate et al. presented a method to P system verification using the Spin model checker [26]. Other approaches have been studied [6,27,19,5] for stochastic systems , where the rules of a P system model of a small gene expression system were checked using a dedicated tool, called PRISM (Probabilistic and Symbolic Model Checker) [22,23]. The work was further extended to incorporate the verification of Spiking Neural P systems [20].
A robust method for formal verification of P systems is represented by model checking. Model checking is a formal verification technique used to demonstrate if a system model satisfies a given specification. This involves that each property of the computer system or biological model is expressed in a formal logic and then verified by systematically exploring all possible states of the model. Model checking explores all possible states and execution paths of the system within its model, eliminating the risk of missing rare corner cases. This constitutes a major advantage over classical testing or system simulation [3]. Moreover, when a property is not satisfied, model checkers generate a counterexample trace that illustrates the sequence of steps leading to failure [25].

As mentioned earlier, PRISM is a probabilistic model checker tool. It supports three different types of probabilistic models: discrete time Markov chains (DTMC), Markov decision processes (MDP) and continuous time Markov chains (CTMC) [5]. Accordingly, its use requires an explicitly constructed formal model. This makes the tool less suitable for systems studied primarily through simulation, where such a specification may not be available. Monte Carlo model checker (MC2) [21] is another tool that performs model checking, analysing simulation traces generated by other tools. MC2 is beneficial for systems with large state spaces, where exhaustive exploration is impractical. Both PRISM and MC2 were part of a structured analysis of model checking tools provided by M. Bakir et al. [4].

Membrane systems have also been successfully applied in robotics. A category of cell-like P systems, called *numerical P systems* has been introduced by Gh. Păun and R. Păun in 2006 [34]. A numerical P system evolves numerical variables using a membrane structure, in the form of a tree. The evolutionary process involves the parallel execution of polynomials with integer coefficients, known as production functions. Each membrane contains one or more production functions. The use of numerical variables inside membranes has made numerical P systems, along with their extension, *enzymatic numerical P systems* (ENPS) [32], efficient for robotic control. Unlike traditional numerical P systems, ENPS employ enzyme variables to dynamically control the activation of rules. The natural modeling of reactive behaviours as well as the inherent parallelism that matches real-time requirements make these systems suitable for robotics.
However, designing controllers for autonomous robots that reliably avoid obstacles is a challenging task, as the relationship between sensor readings and motor commands must be both correct and robust across different environmental conditions. Consequently, ensuring that the resulting controller behaves correctly remains highly delicate. The interaction between the membrane computation, the sensor inputs and optimised weight parameters of the controller must be formally proven to guarantee that the intended obstacle avoidance behaviour is actually produced. Moreover, the above mentioned features of ENPS along with the continuous dynamics of the environmental scenarios create significant verification challenges. Aspects like non-determinism, inherent complexity and the large state space of the model make model checking a rigorous yet computationally feasible approach to verify several temporal properties.

In this paper, we investigate the formal verification of obstacle avoidance robot controllers based on enzymatic numerical P systems. The core component of the controllers is an ENPS model that interacts with the environment and takes sensor values at each simulation step, returning the computed wheel speeds of the robot. As the educational robot used in our studies is actuated by two independent motors, the differences between the speeds serve as the primary mechanism governing obstacle avoidance behaviour.
We show how model checking can be applied to study the probabilistic behaviour of such controllers by verifying several properties defined in MC2 using a logic language called Probabilistic Linear-time Temporal Logic (PLTL). While previous work designed the ENPS-based controllers and evaluated their performance by assessing metrics like traversal time, distance or memory consumption [9], this paper applies model checking to formally quantify their key behavioural properties. Each property captures critical correctness requirements and is classified according to different aspects of system operation. We introduce and describe each of these properties and then explore the results produced by MC2 based on the model and different scenarios that are performed in a dedicated simulation environment, providing an effective way of validating the model.

The paper is structured as follows: Section 2 includes the main concepts to be used in this paper. Section 3 provides an overview of the controllers logic and introduces one enzymatic numerical P system model. Section 4 describes the working methodology, including the tools involved. Section 5 explains the properties used for the formal verification. Section 6 illustrates the outputs of the verification process under different scenarios. In the end, Section 7 states the conclusions and the future work in this area.

## 2 Preliminaries

In this section, we will introduce the principal concepts used in our approach: enzymatic numerical P systems (ENPS), which represent the computational model that is used to build the main logic of the robot controllers and statistical model checking for formal verification.

### 2.1 Enzymatic Numerical P Systems

As stated before, the primary innovation brought by numerical P systems is the use of numerical values inside compartments. In this way, the compartments contain variables instead of objects. Each variable has an initial value that can be an integer or real number. Another important change is made by the *programs*, that are replacing the traditional *communication rules*. At every time unit, a program from each membrane is executed and the variables are evolved, changing their values. The entire system is governed by a global clock operating in discrete time units.

The enzymatic numerical P systems (ENPS) [34,32] are a variant of numerical P systems that use *enzymes* to control the application of a program during the execution. An enzyme is a variable that allows the execution of more than one production function from each membrane. The enzyme dictates if a program is executed or not, based on the following rule: a program can only be applied (is said to be active) if the corresponding enzyme is present in the necessary amount (has a greater value than the minimum between the variables values involved in that program).

The enzymatic numerical P system (EN P system) is defined by the tuple [8]:

$$\Pi = (m, H, \mu, (Var_1, E_1, Pr_1, Var_1(0)), \ldots, (Var_m, E_m, Pr_m, Var_m(0))) \quad (1)$$

where:

- $m \geq 1$ is degree of the system $\Pi$ (the number of membranes);
- $H$ is an alphabet of labels;
- $\mu$ is membrane structure;

- $Var_i$ is a set of variables from membrane $i, 1 \leq i \leq m$;
- $E_i$ is a set of enzyme variables from membrane $i, 1 \leq i \leq m$;
- $Var_i(0)$ is the initial values of the variables from membrane $i, 1 \leq i \leq m$;
- $Pr_i$ is the set of programs from membrane $i, 1 \leq i \leq m$.

The program $Pr_{l_i,i}$, $1 \leq l_i \leq m_i$ can have two forms, the non-enzymatic, which is identical to the one from the classical numerical P systems and the enzymatic form:

i) non-enzymatic

$$F_{l_i,i}(x_{1,i}, \ldots, x_{k,i}) \rightarrow c_{1,i}|v_1 + c_{2,i}|v_2 + \cdots + c_{m_i,i}|v_{m_i} \tag{2}$$

where $F_{l_i,i}(x_{1,i}, \ldots, x_{k,i})$ is the production function, $c_{1,i}|v_1 + c_{2,i}|v_2 + \cdots + c_{m_i,i}|v_{m_i}$ is the repartition protocol, and $x_{1,i}, \ldots, x_{k,i}$ are variables from $Var_i$. Variables $v_1, v_2 \ldots v_{m_i}$ can be from the membrane where the programs are located, and to its upper and inner compartments (membranes), for a particular membrane $i$.

ii) enzymatic

$$F_{l_i,i}(x_{1,i}, \ldots, x_{k,i})|_{e_i} \rightarrow c_{1,i}|v_1 + c_{2,i}|v_2 + \cdots + c_{m_i,i}|v_{m_i} \tag{3}$$

where $e_i$ is an enzymatic variable from $Var_i$, $e_i \notin \{x_{1,i}, \ldots, x_{k,i}, v_1, \ldots, v_{m_i}\}$. The program can be applied at time $t$ only if $e_i > min(x_{1,i}(t), \ldots, x_{k,i}(t))$. All the programs that meet this condition in a membrane will be selected to be executed in parallel.

Let us consider that the system applies a program at time $t \geq 0$. Then, let the computed result of

$$q_{l_i,i}(t) = \frac{F_{l_i,i}(x_{1,i}(t), \ldots, x_{k,i}(t))}{\sum_{j=1}^{n_i} c_{j,i}} \tag{4}$$

represent the *unitary portion* that will be distributed to the variables $v_1, \ldots, v_n$, in accordance with the coefficients $c_{1,i}, \ldots, c_{m_i,i}$, where $c_{j,i} \in \mathbf{N}^+$. Thus, the received values will be $q_{l_i,i}(t) \cdot c_{1,i}, \ldots, q_{l_i,i}(t) \cdot c_{m_i,i}$.

At this point, the previous values of the variables (from time $t-1$) that were present in the production functions are said to be *consumed*, meaning that are set to zero and their new value is the sum of the proportions distributed to variable through the repartition protocols, if they appear in them (on the right side of the program) or remain zero if not.

### 2.2 Statistical Model Checking

Statistical Model Checking (SMC) is a verification technique in which properties are evaluated through repetitive system simulations. It was introduced by H. Younes in 2002 [31]. As the exploration of all the possible states can lead to state explosion problem, in statistical model checking the approximate level of system correctness is generated using statistical methods. Statistical inference can be approached in two main ways: *hypothesis testing* and *estimation*. Hypothesis testing determines the extent to which observations adhere to a known specification, whilst estimation is employed to infer the probable values of parameters, under the assumption that the data has been randomly sampled from a specified distribution [1].
As mentioned earlier, SMC does not analyse the entire state space and generates an estimated quantification of system correctness. Thus, even if it can not guarantee the absolute correctness, it can produce satisfying results when verifying larger models [11,44].

Statistical Model Checking is a scalable verification paradigm, being useful in verification problems for biological and cyber-physical systems (CPS), as the size of the state space can be difficult to be precisely determined (or may be infinite). Typically, the use of SMC in such cases comprises the following steps [2]:

1. Sampling the potential executions (simulations) of the CPS.
2. Specifying the temporal logic formulas using a property specification language.
3. Verifying that each property is valid on the model.
4. Applying a statistical method to estimate the value of the probability of correctness.

A significant limitation is caused by some temporal formulas for which the probability of satisfaction is minimal. The problem with this formulas, called *rare events*, is that the number of simulations required to obtain a reliable estimate becomes infeasible. E.M. Clarke and P. Zuliani proposed an approach to address this problem using Importance Sampling and Cross-entropy method [12].
A wide range of SMC tools are available, each providing different capabilities in terms of performance and execution speed, when addressing verification models of variable size and complexity. In this paper we will use Monte Carlo Model Checker (MC2) to verify the core component of a CPS, more precisely an obstacle avoidance robot controller of an educational robot.

## 3 Robot Controllers

In this section we present the operational logic of the robot controllers based on ENPS models. The controllers are purposed to enable the obstacle avoidance functionality of an educational robot, called E-puck [30]. The robot is equipped

with two motors, one for each wheel. Thus, the wheel speed is dictated by the controller. The obstacle avoidance functionality is achieved using the information retrieved by the proximity sensors. Each sensor is associated with a numerical *weight*, which determines its contribution to the control process and is provided as input to the ENPS model, as will be explained when introducing the model instance. In our experiments, a total of six sensors were employed, arranged in a radial configuration with three sensors placed on each side. The robot also has a GPS module installed in the turret slot to accurately measure coordinates.

### 3.1 Robot Controller Using Static Weights

For a clearer insight into the operational mechanisms of the proposed controller, Algorithm 1 presents the pseudocode representation of the main loop of robot controller using static weight values that are empirically chosen, as defined in [10].

**Algorithm 1** Main loop of the robot controller using static weights

```
1: repeat
2:     for i=1 to number_of_sensors do
3:         sensor_membrane(i) ← value(i)
4:     run one simulation step
5:     read lw, rw from ENPS
6:     leftMotor ← lw
7:     rightMotor ← rw
8: until the end of the road or E-puck goes out of the road
```

One can notice that the above presented pseudocode representation of the algorithm illustrates the way the enzymatic numerical P system (ENPS) exchanges information with the environment. At each simulation step, the sensor values are transmitted to the ENPS using a Python script that implements the logic presented by Algorithm 1 and the communication between the environment and ENPS. In this way, the ENPS takes the sensor values and returns the wheel speeds, denoted by $lw$ and $rw$ (left wheel speed and right wheel speed). The computed values are then assigned to the motor speeds, guiding the robot inside the environment. The tools that facilitate the execution of the ENPS and the simulation of the environment will be introduced in the next section.

### 3.2 Robot Controller Using Dynamically Assigned Weight Values

The controller variant that was presented above is designed for ENPS models employing static weights within the sensor-driven obstacle avoidance process. While this approach simplifies the controller design, ensuring a predictable behaviour, the adaptability of the system to changing environmental conditions

is significantly reduced. More specifically, during a simulation, the same weight values are maintained regardless of the obstacle density or trajectory complexity, employing laborious manual processes for finding appropriate values, that are, in most cases, empirically chosen.
To address this limitation, an adaptive mechanism of assigning weights during the simulation has been introduced in [7]. The robot controller presented in 3.1 has been tuned using a NSGA-II, a multi-objective optimisation algorithm [15]. More precisely, the robot controller interacts with an instance of NSGA-II with two objective functions that is responsible for generating weight values in concordance with the sensor information retrieved from the simulation environment. Algorithm 2 synthesizes the interaction as follows:

**Algorithm 2** Main components of the robot controller using dynamic weights

```
instantiate NSGA-II algorithm
create initial population
repeat
    for i=1 to number_of_sensors do
        sensor_membrane(i) ← value(i)
    run NSGA-II for one generation
    get best individual and update weights
    instantiate new population
    run one simulation step
    read lw, rw from P system
    leftMotor ← lw
    rightMotor ← rw
until the end of the road or E-puck goes out of the road
```

It can be observed that the main components of the controller remain consistent with the implementation illustrated in Algorithm 1, where static weights were employed. In addition, the proximity sensor readings represent the input that is fed to the NSGA-II algorithm. The optimisation process is guided by the objective functions, which evolve a population of candidate weight sets and select, at each control step, the most suitable solution. The resulting weights are subsequently passed to the Enzymatic Numerical P system (ENPS), where membrane programs calculate the wheel velocities, which are then assigned to their corresponding variables (*leftMotor, rightMotor*). The resulting robot behaviour impacts subsequent sensor feedback, forming a continuous *perception–decision–action* loop that facilitates real-time adaptation to dynamic environments.

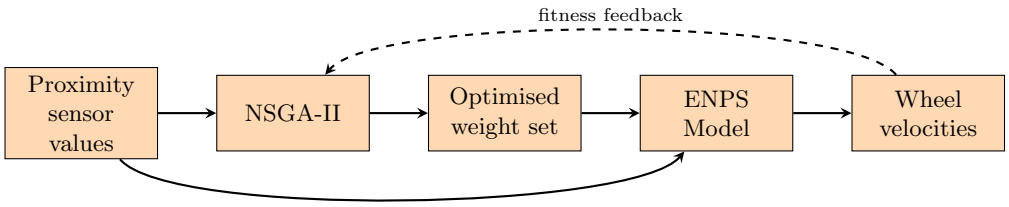



Fig. 1: Dynamic weight optimisation loop integrating NSGA-II with the ENPS-based robot controller

Figure 1 illustrates the information flow of the proposed controller variant. It can be noticed that in the current architecture, the proximity sensor readings serve as input to both the NSGA-II algorithm adaptation, which iteratively constructs the weight set, and the ENPS model, which determines the left and right wheel velocities of the robot.
Although the dynamic assignment of weights demonstrated better performance on certain scenarios and metrics, such as execution time, it also introduced additional complexity into the controller logic. Consequently, the formal verification of the weight assignment requires additional attention to ensure stability, consistency and safe obstacle avoidance. Therefore, this paper also proposes the formal verification of the coordination between the weight parameters and sensor readings.

### 3.3 Enzymatic Numerical P system Model

The enzymatic numerical P system (ENPS) model is the core component of the robot controllers, dictating the path to be followed by adjusting the wheel speeds. The power of ENPS in robot modeling lies in the ability to represent the robot behaviour using membranes and to precisely control the program execution using enzymes. The integration of enzymes in traditional numerical P systems has led to an optimisation of the computation power of robotic systems. Also, the system variables and enzymes can play a central role in transferring the partial values of the outcomes (i.e. speeds).

In our previous work, we defined a methodology that led us to construct different models in a test-driven manner [9]. The objective of the research is to investigate the relationship between the two central components of the model, governing the robot behaviour. More specifically, the study examines how the proximity sensors, radially distributed around the robot and responsible for obstacle detection and their associated weight parameters are influencing the robot trajectories. As all the developed models exhibited strong performance in passing the testing scenarios, due to the simplicity of the membrane structure, only one model is formally presented in this section. Furthermore, the modifications

incorporated into the other developed instances primarily targeted the naturalness of the robot's movements, without altering the relationship between sensors and weights, which constitutes the central focus of this study. As a result, the properties that are defined in this study may hold the same results for all the developed models employing sensor weights. For more details regarding the other model instances, the reader is invited to consult [9].

The model we present in this section is called *Basic Model with Rotation* and it was created after noticing the limitations of the initial model tested in [10]. Building upon this baseline, an additional compartment which is used to compute the product of the travel speed and the sum of the weights has been added. By incorporating this newly defined compartment, the robot behaviour improved. The equations defining the mathematical determination of *weightLeft* and *weightRight*, as well as their contribution to *leftSpeed* and *rightSpeed*, are presented as follows:

$$weightLeft = \sum_{i=1}^{n} weightLeft_i \cdot prox_i \tag{5}$$

$$weightRight = \sum_{i=1}^{n} weightRight_i \cdot prox_i \tag{6}$$

where $prox_i, 1 \leq i \leq n$ represent the sensor values.

In order to proceed with the equations for the left and right wheel speeds, let us first introduce the following function:

$$f(x) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

Then, the motor speeds, denoted by *leftSpeed* and *rightSpeed* are presented as follows:

$$leftSpeed = cruiseSpeed \cdot weightLeft + f(weightLeft) \cdot cruiseSpeed \tag{8}$$

$$rightSpeed = cruiseSpeed \cdot weightRight + f(weightRight) \cdot cruiseSpeed \tag{9}$$

where *cruiseSpeed* represents the predefined velocity of the robot, when moving straight.

The model instance is defined as follows:

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0))) \tag{10}$$

where:

- $m = 3k + 3, k = 6$;
- $H = \{s, w, s_c\} \cup \bigcup_{i=1}^{k} \{c_i, s_i, w_i\}$;
- $\mu = [[[[]_{s_1}[]_{w_1}]_{c_1} \cdots [[]_{s_k}[]_{w_k}]_{c_k}[]_{s_c}]_w]_s$;
- $Var_s = \{x_{s_l}, x_{s_r}\}$, $Var_w = \{x_{w_l}, x_{w_r}, e_w\}$, $Var_{s_c} = \{x_{s_c}\}$,
  $Var_{c_i} = \{x_{c_i,s_l}, x_{c_i,s_r}, x_{c_i,w_l}, x_{c_i,w_r}, e_{c_i}\}, 1 \leq i \leq k$,
  $Var_{s_i} = \{x_{s_i,i}\}, 1 \leq i \leq k$,
  $Var_{w_i} = \{x_{w_i,w_l}, x_{w_i,w_r}, e_{w_i}\}, 1 \leq i \leq k$;
- $Var_i(0) = 0, 1 \leq i \leq m$;
- $Pr_s = \{0 \cdot x_{s_l} \cdot x_{s_r} \rightarrow 1|x_{s_l} + 1|x_{s_r}\}$;
  $Pr_w = \{x_{s_c} \cdot x_{w_l} + f(x_{w_l}) \cdot x_{s_c}|_{e_w} \rightarrow 1|x_{s_l},$
  $x_{s_c} \cdot x_{w_r} + f(x_{w_r}) \cdot x_{s_c}|_{e_w} \rightarrow 1|x_{s_r}\}$;
  $Pr_{s_c} = \{x_{s_c} \rightarrow 1|x_{s_c}\}$;
  $Pr_{c_i} = \{x_{c_i,s_l} \cdot x_{c_i,w_l}|_{e_{c_i}} \rightarrow 1|x_{w_l},$
  $x_{c_i,s_r} \cdot x_{c_i,w_r}|_{e_{c_i}} \rightarrow 1|x_{w_r}\}, 1 \leq i \leq k$;
  $Pr_{s_i} = \{3x_{s_i,i} \rightarrow 1|x_{s_i,i} + 1|x_{c_i,s_l} + 1|x_{c_i,s_r}\},$
  $1 \leq i \leq k$;
  $Pr_{w_i} = \{2x_{w_i,w_l}|_{e_{w_i}} \rightarrow 1|x_{w_i,w_l} + 1|x_{c_i,w_l},$
  $2x_{w_i,w_r}|_{e_{w_i}} \rightarrow 1|x_{w_i,w_r} + 1|x_{c_i,w_r}\}, 1 \leq i \leq k$;

For completeness, the role and interpretation of each variable from the model is clarified below:

- $x_{s_l}$ and $x_{s_r}$ from membrane $s$ represent *leftSpeed* and *rightSpeed*, the sum of the products are accumulated in $s$ after applying the rules from the membranes $c_i$, where $1 \leq i \leq k$;
- $x_{w_l}$ and $x_{w_r}$ from membrane $w$ represent the *weightLeft* and *weightRight*, computed as described in 5 and 6;
- $x_{s_c}$ from membrane $s_c$ is *cruiseSpeed*;
- variables $x_{c_i,s_l}$ and $x_{c_i,s_r}$ from membrane $c_i$ represent the sensor values multiplied inside membranes $s_i$, whilst $x_{c_i,w_l}$, $x_{c_i,w_r}$ are the values of $weightLeft_i$ and $weightRight_i$ derived from membranes $w_i$;
- each pair of weights, ($weightLeft_i$, $weightRight_i$), is denoted by ($x_{w_i,w_l}$, $x_{w_i,w_r}$) and is located in membranes $w_i$, $1 \leq i \leq k$;
- each proximity sensor, $prox_i$, has an associated membrane $s_i$, containing a single variable, $x_{s_i,i}$, $1 \leq i \leq k$, holding the last value reported by the sensor;

One can observe the addition of $x_{s_l}$ and $x_{s_r}$ multiplied by zero in the production function from membrane $s$. The reason for adopting this approach is to represent the reset of the speeds, in order to consume the previous values of *leftSpeed* and *rightSpeed* at each simulation step.

The *weight* membrane $w$ comprises two distinct programs that are responsible of aggregating weight contributions and applying the *cruiseSpeed*. Within the *weight* membrane, the accumulated values of *weightLeft* and *weightRight* are multiplied by *cruiseSpeed*, with a fallback mechanism (adds *cruiseSpeed* if weight

is 0), as outlined in 8 and 9, computing the actual values of the wheel velocities, through the use of enzymes.

It can be noticed that the *cruiseSpeed* membrane $s_c$ incorporates a self-replicating program that preserves the *cruiseSpeed* value and makes it available to the parent compartment $w$. As highlighted by equations 5 and 6, each proximity sensor has a weighted contribution to *weightLeft* and *weightRight*. This contribution is achieved from the compute membranes $c_i, 1 \leq i \leq n$, where $n$ represents the number of sensors. Furthermore, membranes $s_i$ act like a sensor data distribution layer, distributing copies to the corresponding left and right computation variables ($x_{c_i,s_l}$, $x_{c_i,s_r}$) in the parent compartment. An analogous role is assumed by membranes $w_i$ that duplicate each pair of weights, $weightLeft_i$, $weightRight_i$, providing copies ($x_{c_i,w_l}$, $x_{c_i,w_r}$) to the respective compute membrane $c_i$.

## 4 Methodology and Tools

In order to support the verification process, a number of dedicated tools are employed, each fulfilling a different role, from the simulation environment to model checking. In this section we present an overview of the tools used in this study, highlighting their contribution in the verification process. Rather than deploying the controller logic on the physical version of E-puck, we demonstrate its behaviour through a robotic simulation platform that enables the simulation of complex environments, called Webots [29]. As the core component of the controllers is represented by the ENPS model $\Pi$, a tool that is capable of simulating numerical P systems evolutions was needed. Thus, we used PeP [39], a simulator of both classical and enzymatic NPS. Subsequently, we exported the simulation trace produced by the execution of $\Pi$ and analysed it using Monte Carlo Model Checker (MC2) to formally verify some specified properties. In what follows, we provide additional details on the distinct roles fulfilled by each of the tools employed in this process.

### 4.1 Webots

Webots [29,13,28] is a robot simulation software that provides a virtual environment for developing and testing robots in customised environments. Special mechanisms called PROTOS are used in conjunction with default shapes to construct complex environments by adding various objects to the scenes. Webots provides various accessories that can be mapped to robots, such as GPS, light sensors or LED. During the simulation phase, we attached a GPS to the turret slot of the robot in order to pass the system coordinates to the controller.
A simulation scenario is called *world*, defined as a *.wbt* file, containing the definition and placement details for all the object from the scene. The *.wbt* can be edited in order to extend a scene.
A virtual representation of E-puck, equipped with 6 sensors, replicating the main functionalities of the original robot has been involved in each scenario. The robot

controller is editable in the simulation interface and supports multiple programming languages, such as C, Java, Matlab or Python. Our developed obstacle avoidance controller is a Python file that exchanged data with the ENPS model at each simulation step.
For additional details regarding the integration of Webots in a similar experimental setup, the reader is invited to consult [10].

### 4.2 PeP

As already mentioned, the robot controller takes the wheel speed values from the ENPS model $\Pi$ at each simulation step. This process entails the use of a dedicated tool for simulating ENPS.
PeP is an open-source tool developed by A. Florea and C. Buiu [39]. It uses *.pep* input files to model NPS and ENPS instances, by defining the membrane structure, variables, enzymes and rules. PeP provides options for deterministic or randomised program selection and supports generic mathematical functions in production and distribution rules. The tool provides detailed execution traces, illustrating the evolution of variables and enzymes. It can be executed for a predefined number of steps and facilitates options for debug and export.
As the tool demonstrates strong applicability for the experimental evaluation of ENPS models, we used it iteratively inside the main loop of the controller. At the end of each simulation, we exported simulation traces as a CSV file, containing the variable values at each step.
An important advantage of the tool lies in its capability to generate traces in an organised form, representing a foundation for subsequent statistical formal verification using a model checker.

### 4.3 MC2 Model Checker

The Monte Carlo Model Checker (MC2) [21] was developed as a tool for statistical model checking based on simulation. The tool has proven effective results for complex models where exact model checking would be infeasible due to the state-space explosion problem.
MC2 facilitates statistical model checking of simulation traces and supports parallel execution. A notable advantage of the tool is that it does not require a specific modeling language; instead, it is based on simulation traces generated by external tools for both stochastic and deterministic models. The tool is developed in Java and distributed as a *.jar* file.
Properties in MC2 are defined using a logic language called Probabilistic Linear-time Temporal Logic (PLTL). The basic unit of this language is denoted by the Atomic Proposition (AP) which is included in properties without temporal operators. According to the user manual [41], there are five temporal operators defined as follows:

- **Next (X)**: the property must hold true in the next time point;
- **Globally (G)**: the property must hold true always in the future;

- **Finally (F)**: the property must hold true sometime in the future;
- **Until (U)**: the first property must hold true until the second property holds true;
- **Release (R)**: the second property can only ever not hold true if the first property becomes true.

In our approach, we used PeP to export the simulation traces and after minimal format adjustments we passed the file to the model checker. At the same time, we defined several queries written in PLTL into a separate query definition file. Each query has a possible output of true, if the specified property is satisfied in the simulation traces, or false if not. MC2 was selected over other model checking tools (such as PRISM) primarily because it eliminates the need to construct an explicit formal model in a dedicated language. Instead, it operates directly on execution traces generated by external tools. This aligns with our workflow, where the ENPS model is simulated using PeP and the resulting traces are exported for verification. MC2 operates via command-line interface executed from the directory containing the *.jar* file, where we also placed the simulation traces and the file containing the queries. In what follows, we will provide more details about the queries definitions.

## 5 Formal Verification Properties

In this section, several properties are defined using MC2. The main goal of the properties is to verify the critical correctness requirements of the ENPS model $\Pi$ that is the core component of the obstacle avoidance robot controllers. By analysing the formal definition of model $\Pi$, the expected behaviour at each computation step can be formally characterised and expressed using PLTL properties. In addition, model $\Pi$ serves as the analytical foundation from which each property is derived, not as a direct input to the verification tool. The Boolean nature of the expected outcomes (false for conditions that must never be reached, true for conditions that must be reachable) reflects the evaluation of whether the production rules of the ENPS model produce the expected results. Unlike exhaustive model checking approaches, MC2 allows to obtain confidence bounds on property violations without compromising the computational costs. Each property is expressed using Probabilistic Linear-time Temporal Logic (PLTL), employing temporal operators.

To some extent, the properties we present can be classified into the following categories:

- *liveness* properties: ensure that the system does not halt. In our approach, this type of properties verifies that the controller successfully transmits values for the wheel speeds at each simulation step and the system maintains its progress.
- *safety* properties: specify conditions that must always hold true during system execution. In our approach, these properties are related to the controller

capacity to react accordingly to the environmental conditions, i.e. to detect and avoid obstacles. In terms of system states, these properties are specified to guarantee that the verified system does not reach undesirable states, such as getting stuck in an obstacle.
- *invariant* properties: verify that certain behavioural characteristics of the systems remain consistent during the system execution. In our approach, this type of properties ensures that unnatural behaviours, such as spinning moves, do not occur during the simulation.

Observing the below presented properties through simulation is impractical because they specify constraints over multiple states and time points that may not be evident from observing individual execution traces. Thus, applying model checking enables systematic verification of these complex temporal relationships that would otherwise require manual inspection across numerous scenarios. In what follows, we provide both the formal PLTL specification and the description of what each property ensures about controller behaviour, along with the explained possible outputs.

Section 5.1 defines properties that operate on the input provided by the simulation environment to the ENPS (the proximity sensor readings, *s1-s6*) and on the output of the ENPS model (the wheel velocities, *lw* and *rw*). Consequently, they verify the correctness of the sensor-motor mapping regardless of how the weights are obtained. Therefore, they apply equally to both the static and the dynamic controllers that are described in Section 3.

*Remark 1.* For clarity and simplicity, we use the actual variable names from the model implementation rather than introducing the variables defined in the formal definition presented in 3.3. For example, the wheel speeds are denoted by *lw* and *rw* instead of membrane variables $x_{s_l}$ and $x_{s_r}$. For the sensor values, we use $s_i, 1 \leq i \leq 6$ instead of $x_{s_v,i}, 1 \leq i \leq 6$.

### 5.1 Common Properties

*Property 1 (Liveness).* $P =?[F([lw] \neq [rw])]$

**Description**. This property verifies that the controller is capable of producing asymmetric wheel speeds at some point during the simulation, checking the necessary condition for the robot to turn and avoid obstacles. In model $\Pi$, *lw* and *rw* are computed independently by the Weight membrane *w* through two separate rules, each accumulating a distinct weighted sum, *weightLeft* and *weightRight*. The temporal operator "Finally" (F) means that the condition must hold at some point in the future along at least one execution path. By analysing the physical structure of the robot, one can notice that the robot will turn if and only if the wheel speeds will take different values during the simulation. In the context of an obstacle avoidance controller, it is essential to ensure that the wheel speeds are adjusted properly, based on inputs from the environment. This type of property is often encountered in automotive area when verifying the

turning dynamics of the prototype, where asymmetric wheel speeds are expected under certain circumstances.
**Output.** The property will hold true if the wheel speeds will differ at some points during the system execution, ensuring that the controller is able to turn the robot in a direction. If the wheel speeds will maintain equal values during the entire simulation, the model checker will return zero (false), as this means that the robot only moves straight ahead.

*Property 2 (Liveness).* $P =?[F(([lw] \neq [rw]) \wedge X([lw] = [rw]))]$

**Description.** This property verifies that after successfully avoiding an obstacle, the robot eventually reorients itself to a centered trajectory. When the robot avoids an obstacle and sensor values return toward zero, the membranes associated to each proximity sensor forward weighted products to the Weight membrane, causing *weightLeft* and *weightRight* to converge. When both aggregate values evaluate to zero, the fallback mechanism defined in 8 and 9 and implemented through rules in $Pr_w$ produces identical outputs for both wheel speeds, *lw,rw*. Since the Speed membrane resets both wheel speed values at every step, symmetry is restored within a single step, ensuring that no residual values from previous configurations affect subsequent computations. This analysis is useful for identifying undesirable motion patterns, including potential spinning or circular movements, which could compromise the stability and reliability of the navigation. The scope of this property is to determine whether the robot has the ability to return to a stable, centered trajectory after obstacle avoidance.
**Output.** This property will hold true if there exist at least once two consecutive steps in the simulation traces where the speeds are initially different and then they become equal, indicating that the robot follows a straight trajectory after performing an obstacle avoidance manoeuvre.

*Property 3 (Liveness).*
$P =?[F([s1] > 0 \vee [s2] > 0 \vee [s3] > 0 \vee [s4] > 0 \vee [s5] > 0 \vee [s6] > 0)]$

**Description.** This property verifies that model $\Pi$ receives the sensor values from the testing environment. Although the primary aim of system execution verification focuses on the analysis of output values, it is also important to consider and verify the system's interactions with its connected components. As stated in Algorithm 1, in our particular setup, the main loop of the robot controller comprises the exchange of information between the ENPS model and the environment. The desired behaviour implies that $\Pi$ model takes the sensor values from the environment, computes and returns the wheel speeds at each simulation step.
**Output.** The property will hold true if at least one sensor will be triggered at some point in the future. As the absence of nearby obstacles is denoted by $s_i = 0, \forall i \in \{1, \ldots, 6\}$, the use of the Finally (F) operator is appropriate.

*Property 4 (Safety).* $P =?[G([lw] \neq 0 \wedge [rw] \neq 0)]$

**Description.** This property verifies that the wheel speeds remain different from zero throughout the simulation, ensuring continuous motor output at every computation step. As it was defined, the $\Pi$ model computes the values of *lw,rw* wheel speeds that are used in the main loop of the robot controller. Once the simulation starts, the model must return values that move the robot inside the testing environment. The computation is guided by the sensor values and the weights, as stated in $Pr_w$ compartment of the ENPS model. When no obstacle is detected, the *cruiseSpeed* variable value that is persisted by a dedicated program, ensures that both wheel speeds default to the cruise speed value. The "Globally" temporal operator (G) denotes that the property must hold true during the entire execution.
**Output.** The property will hold true if *lw* and *rw* will be different from zero and false if at least one wheel speed becomes zero during the simulation.

*Property 5 (Safety).*

$$P = ?[F((\sum_{i=1}^{3}[s_i] > 0) \wedge (\sum_{i=4}^{6}[s_i] = 0) \wedge X(X(X([lw] \leq [rw])))]$$

**Description.** This property is designed to detect a situation that is not intended during the system execution. An immediate analysis of the robot dynamics indicates that when the proximity sensors detect an obstacle on the left side, the ENPS model $\Pi$ *must* adjust the wheel speeds by producing *lw* strictly greater than *rw*. The property verifies whether there exists at least one state where the sensors placed on the left side are triggered ($s1 + s2 + s3 > 0$), indicating that an obstacle is detected on the left of the robot and *lw* will become less than or equal to *rw* after two simulation steps, which would contradict the intended avoidance behaviour.
One can notice that safety properties were defined as a type of properties that must always hold true during the entire execution of the system. Consequently, the way this definition has been presented may suggest the use of the Globally (G) temporal operator. As it was defined, the G temporal operator asserts that a property must hold at every single state throughout the execution. By using it individually, the G operator operates on individual states rather than sequences of states and does not have the capability to capture consecutive steps or to limit the number of steps over which a property should return true. In this way, the Next operator (X) can be used to verify that a multiple step sequence occurs during execution. By definition, *X(property)* verifies that the property must hold true in the next simulation step. Additionally, *X(X(X(property)))* will determine whether the property is true in the third simulation step (at moment *t+3*). The use of the Next operator in this manner can chain together n-consecutive steps, potentially combined with G or F operators.
Our experiments revealed that having the Globally operator G combined with Next (X) using logical Boolean operators will return false negative outcomes when there are less than n-consecutive steps remaining in the simulation. For example, if the objective is to verify that after detecting an obstacle on the right side, the robot will turn left during the next 3 steps, but just two steps are remaining in the simulation trace, the property will hold false. In order to

avoid these cases, we negate the consequent conditions that must hold true during the next steps and use the Finally (F) operator. This approach avoids the fundamental problem with G enforcing an universal quantification over all states.

With respect to the fourth property definition, the following observations may be drawn:

- multiple conditions are linked using "logical and" (conjunction) operator;
- the Finally operator F incorporates the entire property definition, meaning the returned result will be true if the conditions are met at some point during the simulation;
- by analysing the definition of model $\Pi$, it can be observed that the controller reacts to environmental inputs over a three-step interval. During these steps, the sensor variables are assigned and distributed, the *weightRight* and *weightLeft* values are computed and the motor speeds are determined.

The Finally operator is used to verify whether such situations occur at least once during the simulation. This representation of the property enables the detection of undesired behaviour during simulation, while avoiding the case where the three-step condition cannot be satisfied due to the proximity to the end of the trace (in that case, will correctly return zero, as we cannot determine an undesired behaviour without all the necessary steps).
**Output.** The property will hold true if sometime in the future the sensors on the left side are triggered (and not the others) and the left wheel speed will remain less than or equal to the right wheel speed after two steps. The desired outcome of this property is false, as it checks the presence of an unwanted behaviour.

*Property 6 (Safety).*
$P =?[F((\sum_{i=1}^{3}[s_i] = 0) \wedge (\sum_{i=4}^{6}[s_i] > 0) \wedge X(X(X([lw] \geq [rw]))))]$

**Description.** This property verifies if there exists at least one case when an obstacle is detected on the right side of the robot and the left wheel speed remains greater than or equal to the right wheel speed after two steps. The property implements the above described logic of the fourth property, with the remark that in this case the robot reaction for obstacles positioned on the right side is verified.
**Output.** The property will hold true if there exists at least one point in the execution where the right-sided sensors are triggered and the right wheel will remain less than or equal to the right wheel speed after two consecutive steps. As in the case of the previous property, the desired result of this property is false.

*Property 7 (Invariant).* $P =?[F(([lw] > [rw]) \wedge X([lw] < [rw]))]$

**Description.** This property verifies if the robot exhibits sudden directional changes during the system execution. As in the previous properties, the Next (X) operator is used inside the Finally block, meaning that the property verifies

the presence of two consecutive steps satisfying the conditions in the simulation trace. The ENPS model $\Pi$ returns the wheel speeds at each simulation step, based on the received environment inputs. Consecutive values of $lw$ and $rw$ are determined by *weightLeft* and *weightRight* variables within the Weight membrane $w$. A simulation step constitutes a minimal unit of measurement in the context of a complete execution, being imperceptible in the visual representation of the scene. It is also worth noting that the infrared proximity sensors of E-puck return non-zero readings when an obstacle is within a short detection range of approximately 4 cm. As a result, simultaneous activation of sensors on opposing sides is largely restricted to dense or dynamically changing environments with multiple obstacles. Consistent with our test suite where no such configuration arises, we can assert that in typical obstacle avoidance scenarios, the verified model should not produce two consecutive mutually exclusive outcomes, namely one with the left wheel rotating faster than the right, and the next one with the right exceeding the left one.

**Output.** The property will hold true if there exists at least one case of abrupt directional changes in the simulation trace. As a consistent behaviour implies the robot maintaining its current trajectory without successive abrupt directional changes, the expected output of this property is false in the typical scenarios for which the controller was designed, such as road tests, corridor navigation and arena environments where the robot does not navigate through passages narrow enough to simultaneously trigger sensors on both sides.

### 5.2 Dynamic Weight Assignment

As stated in 3.2, the integration of NSGA-II as an optimisation of the robot controller increases the complexity of the proposed solution. According to this approach, the sensor weights are no longer maintained as fixed parameters. They become dynamic variables whose values continuously evolve according to the current navigation context, which is encoded through sensor values serving as inputs to the optimisation algorithm. However, as the weight values are no longer empirically chosen static parameters, inappropriate weight combinations or abrupt transitions may occur during the simulation. Consequently, additional attention must be dedicated to the formal verification of the weight values in order to ensure stability and safe obstacle avoidance performance. In this regard, several additional properties are proposed to complement the verification conditions previously defined.

*Property 8 (Liveness).* $P =?[F([weightLeft] \neq [weightRight])]$

**Description.** This property verifies that the weight aggregation mechanism of the ENPS model is capable of producing asymmetric total weights across the left and right motor speeds. It aims to find if there exists at least one state in which the accumulated value of *weightLeft* differs from that of *weightRight*. It is worth mentioning that the aggregate weight computation involves the sensor

readings, values different from zero thereby denoting the presence of an obstacle. As can be observed from the membrane structure of model $\Pi$, there are 6 Compute membranes ( $c_i; 1 \leq i \leq 6$). Each of these membranes contributes to *weightLeft* and *weightRight*, as described by the rules from $Pr_{c_i}$, $1 \leq i \leq 6$. Since the accumulation channels operate independently, the model permits, but does not guarantee asymmetric outputs. Consequently, the property verifies that this asymmetry is indeed produced during the execution.

**Output.** This property will hold true if the aggregated weights will differ at least once during the simulation, showing that the model is structurally capable of generating turning behaviour. A result of false would imply that the model always produces symmetric outputs.

*Property 9 (Safety).* $P =?[F((\sum_{i=1}^{3}[s_i] > 0) \wedge (\sum_{i=4}^{6}[s_i] = 0) \wedge X(X([weightLeft] \leq [weightRight])))]$

**Description.** This property validates the correctness of the dynamically generated weights relative to the proximity sensor values. Specifically, it checks if there exists at least one state where the left side proximity sensors *(s1,s2,s3)* report the presence of an obstacle, while no obstacle is detected by the right side sensors and yet the aggregate *weightLeft* is less than or equal to *weightRight* in the subsequent step.

As the robot controller is guided by proximity sensors with associated weights, the operating principle assumes that the weight values influence the obstacle avoidance behaviour. Accordingly, when an obstacle is detected on the left side, the expected behaviour is characterised by positive values for the left weights associated with *(s1,s2,s3)*, exceeding the corresponding right weights. Similarly, an analogous property can be defined for obstacles that are located on the right side by modifying the verification logic to target bigger positive values for the right weights that are associated with *(s4,s5,s6)*, as resulted from equations 5 and 6.

**Output.** This property will hold true if only *(s1,s2,s3)* sensors detect an obstacle, but the aggregate *weightLeft* is less than or equal to *weightRight*. Thus, the expected output of this property is false, as the optimised weight set must produce $weightLeft > weightRight$, when only the left sensors are active. A positive result would indicate that the dynamically assigned weight values do not yield an adequate avoidance response for left-sided obstacle detections.

*Property 10 (Safety).* $P =?[F([s1] > 0 \wedge (\sum_{i=2}^{6}[s_i] = 0) \wedge ([weightLeft1] < 0) \wedge X(X(X([lw] \leq [rw]))))]$

**Description.** This property provides verification of the dynamic weight assignment at the sensor level. A Braitenberg vehicle is a concept from robotics that demonstrates how simple mechanisms incorporating sensors and motors can exhibit intelligent behaviour through mappings between sensory information and motor commands [37]. Inspired by this principle, the scope of this property is

to check whether there exists a state where sensor 1 detects an obstacle on the left side, but the robot does not avoid by turning right, even if no other sensor detects an obstacle, because NSGA-II has generated a negative weight for sensor 1 contribution to the left wheel. In general, left-side obstacles should induce weight configurations that favour $lw > rw$, causing the robot to steer to the right and avoid the obstacle, while right-side obstacles should produce weight configurations that favour $rw > lw$. Using the same verification logic as this property, analogous simulation checks can be conducted for each of the remaining proximity sensors.

**Output.** This property will hold true if there exists at least one point in the execution trace where NSGA-II has generated a negative value for *weightLeft1* that is associated with the detection of an obstacle placed on the left side and the sign error has propagated through model $\Pi$ to produce $lw \leq rw$. Consequently, the expected result of this property is false.

*Property 11 (Safety).*

$$P =?[F(([s1] > 0) \wedge ([weightLeft1] = 0 \wedge [weightRight1] = 0))]$$

**Description.** This property verifies whether the NSGA-II algorithm can produce null weights for proximity sensor s1, while the sensor is activated. Specifically, it queries whether there exists a reachable state where *s1* reports a positive reading, indicating an obstacle in the front left region of the robot and the associated weights, *weightLeft1* and *weightRight1* are simultaneously zero. By looking at equations 5 and 6 and their contribution to 8 and 9, it can be noticed that a correct response to a left-side obstacle implies $weightLeft_i > weightRight_i$, $i \in \{1, 2, 3\}$, whereas effective avoidance of a right-side obstacle requires $weightRight_i > weightLeft_i, i \in \{4, 5, 6\}$. Therefore, with no contribution from the associated weights, the robot will move straight ahead, "ignoring" the reported sensor value.

**Output.** The expected result of this property is 0, meaning that no such state should be reachable under the weight configurations produced by NSGA-II. In this regard, the property can be generalised for all the remaining sensors as the weight generation process is directly targeted, ensuring that the dynamic assignment preserves the desired relationship between sensors placement and the corresponding robot behaviour.

*Property 12 (Invariant).*

$$P =?[F(([weightLeft] < [weightRight]) \wedge X([lw] \geq [rw]))]$$

**Description.** This property checks whether there exists any reachable execution step where *weightLeft* is strictly less than *weightRight* at one step and the left wheel velocity *lw* is greater than or equal to the right wheel velocity *rw* at the next step. It aims to validate the existing mapping between the aggregate weights and wheel velocities.

**Output.** Similarly to some of the previously introduced properties, a value of false for this property indicates the correct functioning of the ENPS based robot controller, as the radial arrangement of the proximity sensors implies that a lower

aggregate left weight must always produce a strictly lower left wheel speed.

*Property 13 (Invariant).*
$P =?[F(([weightLeft] = [weightRight]) \wedge X([lw]! = [rw]))]$

**Description.** This property verifies that equal aggregate weights must always produce equal motor speeds. As mentioned earlier, the simulation traces have highlighted that the absence of nearby obstacles results in computed aggregate weights that remain zero, due to the fact that none of the sensors reports positive values. In such scenarios, the expected behaviour is that the system will produce equal motor speeds, allowing the robot to move in a straight line.
**Output.** This property will hold true if there exists at least one sequence of two simulation steps where aggregate weights produce different motor speeds. The expected value of this property is thereby false, validating the determinism and symmetry of the speed computation with respect to the aggregate weights.

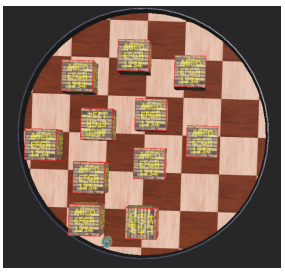
*Property 14 (Invariant).*
$P =?[F(([weightLeft] > [weightRight]) \wedge X([weightRight] > [weightLeft]))]$

**Description.** This property verifies the presence of abrupt changes in weight aggregates. As already mentioned, the weight values are dynamically generated in accordance with the proximity sensor values. Thus, it can be noticed that this property is structurally similar to Property 6. However, whereas Property 6 focuses on detecting abrupt velocity variation, the dynamic generation of weights requires additional verification to ensure that the aggregated weight values do not produce abrupt changes themselves.
**Output.** This property will hold true if abrupt oscillation of aggregate weights is detected. When considered alongside Property 6, the result of this property may suggest the necessity for further tuning of the weight optimisation objectives.

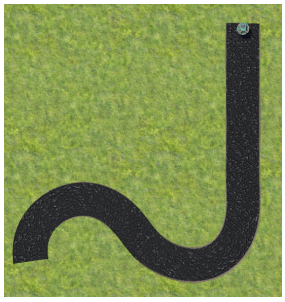
## 6 Results

This section presents the verification results based on the previously described properties. Several simulation scenarios were developed using Webots graphical interface. In what follows, we briefly introduce the scenarios designed to challenge the obstacle avoidance robot controllers based on the ENPS model $\Pi$.
Initially, we generated multiple curved roads based on spines produced using a search-based framework for autonomous systems testing [24,16]. More details regarding the construction of the road tests can be consulted in the previous work [38]. The edges of each road are treated by the robot as obstacles. Therefore, the controller has to guide the robot along the entire road while maintaining a trajectory free of collisions with respect to the boundaries.
Another test scenario consists of situating the robot in circular arenas containing a variable number of obstacles, each being represented within the environment using Webots default shape objects. The arena radius as well as the number of

obstacles are configurable parameters. Given the arrangement of obstacles, the robot's objective is to navigate through the arena without colliding any of them. This scenario can be executed for a predetermined time interval, or until the robot navigates through the obstacles and reaches the boundary of the arena. In our approach, we placed 11 boxes and the simulation is running until the robot escapes the boxes area and reaches the margin of the arena.



Circular arena with obstacles



Curved road

To support reproducibility, the ENPS model, all queries, simulation traces and controller implementations used in this study are available at [40].

Before discussing the results, we emphasize that the verification process was structured into the following stages:

- We simulated the above described scenarios and derived the execution traces of the ENPS model $\Pi$. In the case of curved roads, we extracted traces from 20 roads with different characteristics, whose complexity varied according to the number of curves and their positioning. From the arena scenarios, 5 simulation traces were collected by systematically varying the robot's starting position.
- For the robot controller using dynamically assigned weights, we ran each simulation 10 times, as the optimisation algorithm produces non-deterministic values
- For each testing scenario, the full execution trace was collected, covering the entire simulation from the start until the robot successfully completed the road or reached the boundary of the arena. Each trace contains thousands of entries, where each entry corresponds to a single simulation step. The traces were exported in CSV format by PeP, containing the values of each membrane variable during the simulation. Each row of the CSV corresponds to a simulation step
- The simulation outputs were translated into a data format accepted by the MC2 verification tool.

– Based on the query definition file containing the above presented properties and the simulation output file we performed model checking using MC2 from the Windows command line. MC2 was executed in stochastic mode, evaluating each PLTL property over the set containing 25 simulation traces (20 curved road traces and 5 circular arena traces).

Table 1 presents the content of the query definition file along with the associated results of each property:

Table 1: Verification results

| **ID** | **PLTL Formula** | **Result** |
|---|---|---|
| P1 | $P =?[F([lw] \neq [rw])]$ | 1.0 |
| P2 | $P =?[F(([lw] \neq [rw]) \wedge X([lw] = [rw]))]$ | 1.0 |
| P3 | $P =?[F([s1] > 0 \vee [s2] > 0 \vee [s3] > 0 \vee [s4] > 0 \vee [s5] > 0 \vee [s6] > 0)]$ | 1.0 |
| P4 | $P =?[G([lw] \neq 0 \wedge [rw] \neq 0)]$ | 1.0 |
| P5 | $P =?[F((\sum_{i=1}^{3}[s_i] > 0) \wedge (\sum_{i=4}^{6}[s_i] = 0) \wedge X(X(X([lw] \leq [rw]))))]$ | 0.0 |
| P6 | $P =?[F((\sum_{i=1}^{3}[s_i] = 0) \wedge (\sum_{i=4}^{6}[s_i] > 0) \wedge X(X(X([lw] \geq [rw]))))]$ | 0.0 |
| P7 | $P =?[F(([lw] > [rw]) \wedge X([lw] < [rw]))]$ | 0.0 |
| P8 | $P =?[F([weightLeft] \neq [weightRight])]$ | 1.0 |
| P9 | $P =?[F(((\sum_{i=1}^{3}[s_i] > 0) \wedge (\sum_{i=4}^{6}[s_i] = 0)) \wedge X(X([weightLeft] \leq [weightRight]))))]$ | 0.0 |
| P10 | $P =?[F(([s1] > 0) \wedge (\sum_{i=2}^{6}[s_i] = 0) \wedge ([weightLeft1] < 0) \wedge X(X(X([lw] \leq [rw])))))]$ | 0.0 |
| P11 | $P =?[F(([s1] > 0) \wedge ([weightLeft1] = 0 \wedge [weightRight1] = 0))]$ | 0.0 |
| P12 | $P =?[F(([weightLeft] < [weightRight]) \wedge X([lw] \geq [rw]))]$ | 0.0 |
| P13 | $P =?[F(([weightLeft] = [weightRight]) \wedge X([lw]! = [rw]))]$ | 0.0 |
| P14 | $P =?[F(([weightLeft] > [weightRight]) \wedge X([weightRight] > [weightLeft]))]$ | 0.0 |

Table 1 illustrates the output from MC2 for the model checking of robot controllers based on model $\Pi$. The results of 0.0 and 1.0 indicate that each property was either satisfied or violated consistently across all traces. This uniformity is expected, as the verified properties target the structural correctness of the ENPS model itself rather than scenario-dependent behaviour. Specifically, they evaluate the rules governing weight aggregation, sensor-motor mapping and wheel velocity computation. The query results validate that model $\Pi$ ensures the robot movement while simultaneously exhibiting the capability to adjust the wheel speeds as dictated by the environmental dynamics (transmitted as sensor values with associated static/dynamic weights) and computed using the kinetic rules implemented as membrane programs. All the temporal properties evaluated through MC2 model checking returned the desired outcome, demonstrating alignment between the observed system behaviour and the expected functional requirements. While the consistent results across all tested scenarios support the validity of the findings, statistical model checking estimates property satisfaction

based on sampled executions and does not provide guarantees beyond the use case scenarios provided and the diversity of the collected trace set.

## 7 Conclusions and Future Work

This paper proposes a practical approach for verifying two similar obstacle avoidance robot controllers based on ENPS, one using static weights and the other one taking dynamic weights returned by an integrated multi-objective optimisation algorithm. The verification process involved the use of several tools, with their selection and roles described within this paper. The use of statistical model checking, rather than exhaustive testing, enables the derivation of confidence bounds without compromising computational efficiency.

The results of our experiments proved that the ENPS model correctly implements the desired control logic using both empirical static weights and the weight configurations produced by the NSGA-II adaptation. Furthermore, several properties demonstrated that, within the simulation traces, the integration of the multi-objective optimisation algorithm within the main logic of the controller does not introduce inconsistencies in the mapping from proximity sensor readings to wheel velocities, thus serving as an argument in validating the generated weight sets.

As future work, the proposed approach will be extended with properties that depend not only on the ENPS model but also on the execution scenario. Unlike the properties considered in this study, which verify model correctness by producing deterministic binary outcomes, the new properties will be evaluated over collections of simulation traces, enabling the estimation of probabilities associated with scenario-specific events. For example, given a set of traces collected from different road tests, MC2 can estimate the probability that the robot first detects an obstacle on its right side.

It is worth mentioning that ENPS was adopted in this study because it constitutes the core modelling component of the robot controller. However, the above presented experimental stages can lead to a generalised approach for the formal verification of different numerical P systems variants (e.g., employing Boolean conditions) using model checking.

## References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Transactions on Modeling and Computer Simulation (TOMACS) **28**(1), 1–39 (2018)
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
3. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipate, F.: High performance simulations of kernel P systems. In: 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS). pp. 409–412. IEEE (2014)

4. Bakir, M.E., Gheorghe, M., Konur, S., Stannett, M.: Comparative analysis of statistical model checking tools. In: International Conference on Membrane Computing. pp. 119–135. Springer (2016)
5. Bernardini, F., Gheorghe, M., Romero-Campero, F.J., Walkinshaw, N.: A hybrid approach to modeling biological systems. In: International Workshop on Membrane Computing. pp. 138–159. Springer (2007)
6. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F.J., Krasnogor, N., Gheorghe, M.: Infobiotics Workbench: A P systems Based Tool for Systems and Synthetic Biology, pp. 1–41. Springer International Publishing, Cham (2014)
7. Bobe, R.T., Gheorghe, M., Ipate, F.: Submitted: Sensor-driven evolutionary optimisation of mobile robot controllers using enzymatic numerical P systems (2026), https://doi.org/10.13140/RG.2.2.23913.04969, researchGate [Preprint]
8. Bobe, R.T., Gheorghe, M., Ipate, F., Niculescu, I.M.: Numerical P systems: Variants and applications (2024)
9. Bobe, R.T., Gheorghe, M., Ipate, F., Niculescu, I.M.: A test-driven methodology for designing robot controllers and simulators using enzymatic numerical P system models (2024)
10. Bobe, R.T., Ipate, F., Niculescu, I.M.: Modelling and search-based testing of robot controllers using enzymatic numerical P systems. In: Cheval, H., Leuştean, L., Sipoş, A. (eds.) Proceedings 7th Symposium on Working Formal Methods, Bucharest, Romania, 21-22 September 2023. Electronic Proceedings in Theoretical Computer Science, vol. 389, pp. 1–10. Open Publishing Association (2023). https://doi.org/10.4204/EPTCS.389.1
11. Buchholz, P.: A new approach combining simulation and randomization for the analysis of large continuous time markov chains. ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**(2), 194–222 (1998)
12. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: International symposium on automated technology for verification and analysis. pp. 1–12. Springer (2011)
13. Couceiro, M.S., Vargas, P.A., Rocha, R.P.: Bridging the reality gap between the webots simulator and e-puck robots. Robotics and Autonomous Systems **62**(10), 1549–1567 (2014). https://doi.org/10.1016/j.robot.2014.05.007
14. de la Cruz, R.T.A., Cabarle, F.G.C., Macababayao, I.C.H., Adorna, H.N., Zeng, X.: Homogeneous spiking neural P systems with structural plasticity. Journal of Membrane Computing **3**(1), 10–21 (2021)
15. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE transactions on evolutionary computation **6**(2), 182–197 (2002). https://doi.org/10.1109/4235.996017
16. Gambi, A., Jahangirova, G., Riccio, V., Zampetti, F.: Sbst tool competition 2022. In: Proceedings of the 15th Workshop on Search-Based Software Testing. pp. 25–32 (2022). https://doi.org/10.1145/3526072.3527538
17. Gheorghe, M., Ceterchi, R., Ipate, F., Konur, S., Lefticaru, R.: Kernel P systems: from modelling to verification and testing. Theoretical Computer Science **724**, 45–60 (2018)
18. Gheorghe, M., Ipate, F., Lefticaru, R., Dragomir, C.: An integrated approach to P systems formal verification. In: International Conference on Membrane Computing. pp. 226–239. Springer (2010)
19. Gheorghe, M., Konur, S., Ipate, F.: Kernel P systems and stochastic P systems for modelling and formal verification of genetic logic gates. In: Advances in Unconventional Computing: Volume 1: Theory, pp. 661–675. Springer (2016)

20. Gheorghe, M., Lefticaru, R., Konur, S., Niculescu, I.M., Adorna, H.N.: Spiking neural P systems: matrix representation and formal verification. Journal of Membrane Computing **3**(2), 133–148 (2021)
21. Gilbert, D., Donaldson, R.: A monte carlo model checker for probabilistic ltl with numerical constraints (2008)
22. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. Theoretical Computer Science **391**(3), 239–257 (2008)
23. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: Prism: A tool for automatic verification of probabilistic systems. In: International conference on tools and algorithms for the construction and analysis of systems. pp. 441–444. Springer (2006)
24. Humeniuk, D., Antoniol, G., Khomh, F.: Ambiegen tool at the sbst 2022 tool competition. In: Proceedings of the 15th Workshop on Search-Based Software Testing. pp. 43–46 (2022). https://doi.org/10.1145/3526072.3527531
25. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press (2004)
26. Ipate, F., Lefticaru, R., Tudose, C.: Formal verification of P systems using spin. International Journal of Foundations of Computer Science **22**(01), 133–142 (2011)
27. Konur, S., Gheorghe, M.: A property-driven methodology for formal analysis of synthetic biology systems. IEEE/ACM Transactions on Computational Biology and Bioinformatics **12**(2), 360–371 (2014)
28. Ma'arif, A., Nuryono, A.A., et al.: Vision-based line following robot in webots. In: 2020 FORTEI-International Conference on Electrical Engineering (FORTEI-ICEE). pp. 24–28. IEEE (2020)
29. Michel, O.: Cyberbotics ltd. webots™: professional mobile robot simulation. International Journal of Advanced Robotic Systems **1**(1), 5 (2004). https://doi.org/10.5772/5618
30. Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klaptocz, A., Magnenat, S., Zufferey, J.C., Floreano, D., Martinoli, A.: The e-puck, a robot designed for education in engineering. In: Proceedings of the 9th conference on autonomous robot systems and competitions. pp. 59–65. IPCB: Instituto Politécnico de Castelo Branco (2009)
31. Musliner, D.J., Younes, H.L.: Probabilistic plan verification through acceptance sampling
32. Pavel, A., Arsene, O., Buiu, C.: Enzymatic numerical P systems-a new class of membrane computing systems. In: 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA). pp. 1331–1336. IEEE (2010). https://doi.org/10.1109/BICTA.2010.5645071
33. Păun, G.: Membrane computing: an introduction. Springer Science & Business Media (2002). https://doi.org/10.1007/978-3-642-56196-2
34. Păun, G., Păun, R.: Membrane computing and economics: Numerical P systems. Fundamenta Informaticae **73**(1-2), 213–227 (2006), http://content.iospress.com/articles/fundamenta-informaticae/fi73-1-2-20
35. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
36. Qiu, C., Xue, J., Liu, X., Li, Q.: Deep dynamic spiking neural P systems with applications in organ segmentation. Journal of Membrane Computing **4**(4), 329–340 (2022)
37. Shaikh, D., Rañó, I.: Braitenberg vehicles as computational tools for research in neuroscience. Frontiers in bioengineering and biotechnology **8**, 565963 (2020)

38. Traian Bobe, R., Gheorghe, M., Ipate, F., Mihai Niculescu, I.: A test-driven methodology for designing robot controllers and simulators using enzymatic numerical P system models. SIMULATION p. 00375497241292347 (2024)
39. A. G. Florea and C. Buiu: PeP - an open-source software simulator of Numerical P systems and Numerical P systems with Enzymes 2017. [Online]. Available: https://github.com/andrei91ro/pep
40. Github simulation results folder https://github.com/radubobe/Mobile-Robots-Optimisation/tree/master/ICMC%20MC2%20Verification
41. MC2 User Manual, https://people.brunel.ac.uk/~csstdrg/courses/glasgow_courses/website_sysbiomres/software/mc2/UserManual.pdf
42. Valencia-Cabrera, L., Song, B.: Tissue P systems with promoter simulation with mecosim and p-lingua framework. Journal of Membrane Computing **2**(2), 95–107 (2020). https://doi.org/10.1007/s41965-020-00037-z
43. Verlan, S., Freund, R., Alhazov, A., Ivanov, S., Pan, L.: A formal framework for spiking neural P systems. Journal of Membrane Computing **2**(4), 355–368 (2020)
44. Younes, H.L., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. International Journal on Software Tools for Technology Transfer **8**(3), 216–228 (2006)
45. Yu, W., Wu, J., Chen, Y., Wu, Y.: Fuzzy tissue-like P systems with promoters and their application in power coordinated control of microgrid. Journal of Membrane Computing **5**(1), 1–11 (2023). https://doi.org/10.1007/s41965-022-00109-2
46. Zhang, G., Pérez-Jiménez, M.J., Gheorghe, M.: Real-life applications with membrane computing. Springer (2017). https://doi.org/10.1007/s00500-017-2906-y

# Reducing capacity in P Colonies

Lucie Ciencialová[1][0000−0002−0877−7063] and Luděk Cienciala[1][0000−0001−9473−5945]

Institute of Computer Science, Faculty of Philosophy and Science in Opava, Silesian University in Opava, Opava, Czech Republic
{lucie.ciencialova,ludek.cienciala}@fpf.slu.cz

**Abstract.** P colonies are distributed computational models consisting of simple agents interacting through a shared environment. An important parameter of the model is the capacity, i.e., the number of objects stored by each agent. In this paper, we investigate the reduction of capacity from 2 to 1 for one-agent P colonies without checking rules. We present a constructive transformation that replaces a capacity 2 agent by a fixed set of capacity 1 agents while preserving the behaviour of the original system. The construction demonstrates that the inability of a capacity 1 agent to execute two rules simultaneously can be overcome by coordinating a finite number of auxiliary agents. As a result, every one-agent P colony of capacity 2 without checking rules can be simulated by an equivalent capacity 1 P colony consisting of 43 agents.

**Keywords:** P colonies · Capacity reduction · P colonies without checking rules · Membrane computing

## 1 Introduction

P colonies represent a class of distributed computational models inspired by the behaviour of colonies of simple agents. Each agent has a limited internal memory represented by a fixed number of objects and cooperates with the environment by applying simple evolution and communication rules. Despite their structural simplicity, P colonies are able to achieve considerable computational power, including computational completeness in several restricted variants.

An important parameter influencing the computational capabilities of a P colony is its capacity, i.e., the number of objects stored inside each agent. The original formulation of P colonies was based on capacity 2, and many results concerning computational completeness were obtained for this setting. The question whether the capacity can be reduced while preserving the computational power is therefore a natural research problem.

In this paper, we investigate the reduction of capacity from 2 to 1 for one-agent P colonies without checking rules. The main difficulty of such a reduction is caused by the fact that a capacity 2 agent can simultaneously store two objects and apply a program consisting of two rules, while a capacity 1 agent has only a single object available for representing its current state. We show that

this limitation can be overcome by introducing additional auxiliary agents that simulate the behaviour of the original agent.

The presented construction is motivated by the result from [6], where it was shown that

$$NPCOL_{par/seq}(2, 1, *) = NMAT.$$

Since the considered systems contain only one agent, the sequential and parallel derivation modes coincide. This result shows that one-agent capacity 2 P colonies without checking rules have exactly the power of matrix grammars. Consequently, it provides an important reference point for capacity reduction: the computational power obtained for one-agent capacity 2 systems can be characterized precisely, and the presented construction complements this result by showing that the same behaviour can be simulated using capacity 1 agents.

The remainder of the paper is organized as follows. Section 2 recalls the basic notions used throughout the paper, including multisets and matrix grammars. Section 3 introduces P colonies and summarizes selected results concerning their computational power. Section 4 presents the construction reducing the capacity of one-agent P colonies from 2 to 1 and proves its correctness.

## 2 Preliminaries

In this section, we recall basic notions used in the paper. We assume that the reader is familiar with formal language theory. We briefly recall the notions of multisets and matrix grammars.

A multiset over a finite alphabet $A$ is a mapping $M : A \to \mathbb{N}$ assigning to each object $a \in A$ its multiplicity in $M$. The set of all multisets over $A$ is denoted by $A^{\#}$. The multiplicity of an object $a$ in a multiset $M$ is denoted by $M(a)$. A multiset can also be represented by a string in which the number of occurrences of every symbol corresponds to its multiplicity.

For two multisets $M_1$ and $M_2$, the sum $M_1 + M_2$ and the difference $M_1 - M_2$ are defined in the usual way. The difference is considered only if $M_2(a) \leq M_1(a)$ for every $a \in A$. We say that $M_1$ is contained in $M_2$, denoted by $M_1 \subseteq M_2$, if $M_1(a) \leq M_2(a)$ for every $a \in A$.

A multiset is often used to describe the content of a computational unit. In particular, the state of an agent in a P colony is represented by a multiset of objects. Since the capacity of an agent is fixed, only multisets of a given cardinality are considered.

### 2.1 Matrix Grammars

We briefly recall the notion of context-free matrix grammars without appearance checking, following [6].

A context-free matrix grammar is a quadruple $G = (N, T, S, M)$, where $N$ and $T$ are finite disjoint sets of nonterminal and terminal symbols, respectively, $S \in N$ is the start symbol, and $M$ is a finite set of matrices. Each matrix is a finite sequence of context-free productions over $N \cup T$.

Let $m = (p_1, \ldots, p_k) \in M$. We write $u \Rightarrow_m v$ if there exist strings $w_0, w_1, \ldots, w_k \in (N \cup T)^*$ such that $w_0 = u$, $w_k = v$, and $w_i$ is obtained from $w_{i-1}$ by applying the production $p_i$, for $1 \leq i \leq k$. Thus, a derivation step consists of applying all productions of the selected matrix in the prescribed order.

The language generated by $G$, denoted by $L(G)$, consists of all terminal strings derivable from the start symbol by successive applications of matrices.

The family of languages generated by context-free matrix grammars without appearance checking is denoted by $MAT$. Since P colonies generate sets of natural numbers rather than languages, we consider Parikh images of these languages. The family of all Parikh images of languages from $MAT$ is denoted by $PsMAT$. Throughout this paper, we use the notation $NMAT$ for the corresponding family of sets of natural numbers.

Without loss of generality, matrix grammars can be assumed to be in binary normal form. In this normal form, the nonterminal alphabet is partitioned into $N = N_1 \cup N_2 \cup \{S, f\}$, and matrices are restricted to four types: an initial matrix $(S \to XA)$, production matrices $(X \to Y,\ A \to x)$, terminal matrices $(X \to f,\ A \to x)$, where $|x| \leq 2$, and a unique final matrix $(f \to \lambda)$. Moreover, there is exactly one initial matrix and one final matrix, the latter being applied only in the last step of a successful derivation.

### 2.2 P colonies

The notion of a P colony was originally introduced in [8] and subsequently extended and formalized in [7, 4].

**Definition 1.** *A P colony of capacity $k$, where $k \geq 1$, is a construct*
$$\Pi = (A, e, f, v_E, B_1, \ldots, B_n), \text{ where}$$

- *$A$ is a finite alphabet whose elements are called objects;*
- *$e \in A$ denotes the basic object (also called the environmental object) of the colony;*
- *$f \in A$ is the final object used for representing the output of the computation;*
- *$v_E$ is a finite multiset over $A - \{e\}$ representing the initial content of the environment;*
- *$B_i$, $1 \leq i \leq n$, are agents. Each agent $B_i = (o_i, P_i)$ consists of:*
  - *$o_i$, a multiset over $A$ containing exactly $k$ objects, which specifies the initial state of the agent;*
  - *$P_i = \{p_{i,1}, \ldots, p_{i,k_i}\}$, a finite set of programs. Every program contains exactly $k$ rules, where each rule is of one of the following forms:*
    - *$a \to b$, where $a, b \in A$, called an evolution rule;*
    - *$c \leftrightarrow d$, where $c, d \in A$, called a communication rule;*
    - *$r_1/r_2$, called a checking rule, where both $r_1$ and $r_2$ are evolution rules or both are communication rules.*

Before describing the computational behaviour of a P colony, we provide some additional explanations of its components.

Throughout the paper, we consider the expression "*object a inside agent B*" and the statement "$a \in w$, *where* $w$ *is the current state of agent* $B$" to have the same meaning.

The first class of rules appearing in agent programs are *evolution rules*, written in the form $a \to b$. Applying such a rule replaces an object $a$ located inside an agent by an object $b$.

The second class consists of *communication rules* of the form $c \leftrightarrow d$. Their application enables an exchange of objects between an agent and the environment: the object $c$ from the agent is moved to the environment, while the object $d$ from the environment is transferred into the agent.

The last type of rules are *checking rules*. A checking rule combines two rules of the same type, either two evolution rules or two communication rules. In the case of applying a checking rule $r_1/r_2$, the rule $r_1$ has priority over $r_2$. Therefore, the agent first verifies whether $r_1$ is applicable. If this is the case, $r_1$ must be executed; otherwise, the alternative rule $r_2$ is used.

The three mentioned rule types represent the basic mechanisms of P colonies. However, several extensions and modifications introducing additional rule types have been investigated in different variants of the model. These extensions will be mentioned in subsequent sections.

The behaviour of an agent is determined by its programs. By executing a program, an agent can modify its own state and, in the case of communication rules, also influence the state of the environment.

The environment contains a finite number of copies of non-environmental objects (possibly zero) and an unlimited supply of the environmental object $e$.

During the execution of a program, every object contained in the corresponding agent is processed according to the rules of the program. Consequently, depending on the applied rules, the execution can modify not only the agent itself but also the environment. This mutual interaction between agents and the environment represents the fundamental principle underlying the operation of P colonies.

The computation of a P colony starts from its initial configuration.

The *initial configuration* is an $(n+1)$-tuple consisting of the initial contents of all agents and the environment. It is determined by the multisets $o_i$, $1 \leq i \leq n$, and by the multiset $v_E$. More generally, a configuration of a P colony $\Pi$ is represented as $(w_1, \ldots, w_n, w_E)$, where $|w_i| = k$ for $1 \leq i \leq n$, $w_i$ denotes the current contents of the $i$-th agent, and $w_E \in (A - \{e\})^*$ represents the objects currently present in the environment except for the object $e$.

During each *computation step* (transition), the states of agents and the environment are updated according to the selected programs. In the *maximally parallel* derivation mode, every agent having at least one applicable program chooses one of them nondeterministically and executes it. In contrast, in the *sequential* derivation mode, only a single agent can execute one of its applicable programs in a given step. If several programs are available for an agent, the choice among them is nondeterministic.

A finite sequence of transitions forms a *computation*. Such a computation is called halting if it reaches a configuration in which no program can be applied by any agent. The result associated with a halting computation is defined as the number of occurrences of the final object $f$ in the environment of the final configuration.

Since the selection of applicable programs is nondeterministic, one initial configuration can lead to multiple different computations. Therefore, every P colony determines a set of numbers, denoted by $N(\Pi)$, consisting of all results obtained from its possible halting computations.

In the original formulation of the model [8], each agent contained exactly two objects and each program consisted of only two rules. Additionally, the initial configuration was fixed as $(ee, \ldots, ee, \varepsilon)$, meaning that the environment did not contain any input objects at the beginning of the computation.

The number of agents in a P colony is called its degree. The maximal number of programs assigned to a single agent is referred to as the height, while the number of objects contained in an agent determines the capacity of the colony. The family of all sets of numbers $N(\Pi)$ generated by P colonies with capacity at most $c \geq 0$, degree at most $n \geq 0$, and height at most $h \geq 0$, using checking programs and operating in the sequential mode, is denoted by $NPCOL_{seq}K(c, n, h)$. The corresponding family for P colonies operating in the maximally parallel mode is denoted by $NPCOL_{par}K(c, n, h)$. If the parameters $n$ or $h$ are not restricted, they are replaced by the symbol $*$. For P colonies whose programs do not contain checking rules, the symbol $K$ is omitted.

Despite their structural simplicity, P colonies possess considerable computational capabilities due to their distributed architecture and, especially, their parallel mode of operation. In many cases, computational completeness can be achieved even with a small number of agents and limited program structures. This section summarizes selected results concerning the computational power of P colonies. Most of these results are obtained by constructing simulations of register machines, which also provides insight into the computational principles of these classical models.

### 2.3 Computational Power

Although P colonies are based on a very simple distributed computational mechanism, they exhibit a high computational power. In particular, several variants without checking rules have been shown to reach the computational power of register machines.

For restricted P colonies, where each program consists of one evolution rule and one communication rule, the following results are known:

- $NPCOL_{par}R(2, *, 5) = NRE$ in [6],
- $NPCOL_{par}R(2, 2, *) = NRE$ in [2].

These results demonstrate that checking rules are not necessary for obtaining computational completeness when parallel cooperation of agents is available.

An important result concerning the influence of the number of agents was presented in [6]. It was shown that

$$NPCOL_{par/seq}(2, 1, *) = NMAT.$$

Since only one agent is considered, the parallel and sequential modes coincide. This result is significant because it shows that reducing the degree of the colony to one while keeping the capacity equal to two leads to a model with limited computational power, namely the power of matrix grammars.

Further results confirm that computational completeness can also be achieved for other restricted variants without checking rules. In particular,

- $NPCOL_{par}H(3, 2, *) = NRE$ in [1],
- $NPCOL_{par}(1, 6, *) = NRE$ in [3],
- $NPCOL_{par}(2, 35, 8) = NRE$, $NPCOL_{par}R(2, 57, 8) = NRE$, and $NPCOL_{par}(3, 35, 7) = NRE$ in [5].

## 3 Reducing the Capacity of P colonies

In this section, we investigate the possibility of reducing the capacity of P colonies while preserving their computational power. First, we focus on the transformation of P colonies of capacity 2 into equivalent P colonies of capacity 1. The presented construction is given for one-agent P colonies without checking rules. The main idea is to replace the parallel application of two rules stored in a single program by a sequence of auxiliary computations performed by several capacity 1 agents.

**Theorem 1.** *For every one-agent P colony of capacity 2 without checking rules there exists an equivalent P colony of capacity 1 without checking rules with 43 agents.*

*Proof.* Let $\Pi = (A, e, f, v_E, B)$ be a one-agent P colony of capacity 2 without checking rules, where $B = (o, P)$, $o$ is the initial multiset of the agent containing two objects, and $P = \{p_1, p_2, \ldots, p_h\}$ is the finite set of programs of the agent.

From $\Pi$ we construct a P colony $\Pi' = (A', e, f, v'_E, B'_1, \ldots, B'_{43})$ of capacity 1 without checking rules. The number of agents is fixed and equal to 43, independently of the original system. However, the size of the alphabet $A'$ and the total number of programs of $\Pi'$ depend on the cardinality of the alphabet $A$ and on the number $h$ of programs of $\Pi$.

The construction is presented in two parts. First, we introduce all objects used by the constructed P colony, including the auxiliary objects required during the simulation. Afterwards, we give an informal description of how one computation step of $\Pi$ is simulated by $\Pi'$. The complete specification of all agents and their programs is then given in the remainder of the proof.

The alphabet $A'$ contains all objects of the original alphabet $A$. Moreover, for every pair of objects $a, b \in A$, the alphabet contains a new object $\boxed{ab}$

representing the pair of objects currently stored in the original agent. Hence, $A' \supseteq A \cup \{\boxed{ab} \mid a, b \in A\}$.

To define the remaining objects, we first assign a unique identifier to each program of $\Pi$. Thus, every program is identified by a distinct natural number. For every program $r$, the alphabet contains the objects $r$, $r'$, $r''$, $r'''$, $r_{ok}$, $r'_{ok}$, $r_{ko1}$, $r'_{ko1}$, $r''_{ko1}$, $r_{ko2}$, $r''_{ko2}$, $r_{ko3}$, $D_r$, $d_r$, $D'_r$, $\textcircled{r}$, $\boxed{r}$, $D_{rko1}$, $D_{rko2}$, $D_{r'}$, $D'_{r'}$, $\textcircled{r'}$, $r_0$, $r_1$, $r_2$, $r'_2$, $r_3$, and $r_4$. Additional objects derived from the program identifiers will be introduced later together with the corresponding parts of the construction.

Finally, the alphabet contains the auxiliary objects $H$, $H_1$, $d$, $d'$, $d''$, $d'''$, $d^{iv}$, $d^v$, $\overline{d}$, $\overline{d'}$, and $\overline{d_4}$, which do not belong to the original alphabet $A$.

The construction is based on the fact that the original P colony contains only one agent. Consequently, the applicability of its programs depends solely on the current content of the agent and on the availability of the required objects in the environment. Since no other agents are present, communication rules are never affected by concurrent program selections of other agents.

Instead of determining the set of applicable programs with respect to the current environment, we first consider only the content of the agent. For every pair of objects stored in the agent, we determine the set of programs whose applicability is not excluded by the agent content itself; the presence of the required environmental objects is ignored at this stage. We call such programs *blindly applicable*.

Since each program has a unique identifier, the set of blindly applicable programs can be ordered according to these identifiers. This allows us to define a mapping that assigns to every object $\boxed{ab}$, where $a, b \in A$, the ordered sequence of identifiers of all blindly applicable programs. These sequences constitute the basis for introducing further auxiliary objects of the alphabet $A'$.

Let $r_1, r_2, \ldots, r_k$ be such a sequence, denoted by $list$. The alphabet $A'$ contains the object $\boxed{|list}$. Moreover, for every $r_i \in list$, we introduce the object $\boxed{r_i|list_i}$, where $list_i$ is the sequence obtained from $list$ by removing the element $r_i$.

For every sequence $list_i$ obtained in this way, we introduce the corresponding object $\boxed{|list_i}$ into $A'$. We also introduce further objects obtained by removing one element $r$ from a given sequence. If the sequence contains only the element $r$, the removal produces the object $\boxed{r|\emptyset}$. If the sequence is empty, the construction terminates with the object $\boxed{\emptyset|\emptyset}$. Thus, the recursive construction of shortened sequences and corresponding objects ends.

For every object $\boxed{r|list}$, the alphabet $A'$ additionally contains the objects $\boxed{r|list}'$, $\boxed{r|list}''$, $\boxed{r|list}^{ko}$, $\boxed{r|list}^{ok}$ and $\boxed{r|list}_i; 1 \leq i \leq 4$.

We now give an informal description of the individual phases of the simulation of one computation step of the P colony $\Pi$ by the constructed P colony $\Pi'$. After the informal description of each phase, we specify the programs of the agents that are applied during the corresponding phase.

The first we define mapping $\rho$ Let $I_P = \{1, 2, \ldots, h\}$ be the set of identifiers of programs of the original agent. We define the set of all possible program lists as $\mathcal{S}_P = \{(r_1, r_2, \ldots, r_k) \mid 0 \leq k \leq h,\ r_i \in I_P,\ r_i \neq r_j \text{ for } i \neq j\}$.

We define a mapping

$$\varphi : \{\boxed{ab} \mid a, b \in A\} \to \mathcal{S}_P$$

as follows. For every pair of objects $a, b \in A$, let $BA(\boxed{ab})$ be the set of identifiers of all blindly applicable programs for the agent content $\boxed{ab}$. If $BA(\boxed{ab}) = \{r_1, r_2, \ldots, r_k\}$ and $r_1 < r_2 < \cdots < r_k$, then $\varphi(\boxed{ab}) = (r_1, r_2, \ldots, r_k)$. If no program is blindly applicable for $\boxed{ab}$, then $\varphi(\boxed{ab}) = \emptyset$.

Agent $B_1'$ corresponds to the original agent of the P colony $\Pi$. At the beginning of the computation, it contains an object representing the pair of objects stored in the initial configuration of the original agent $B$. Let the content of the original agent $B$ be $ab$. Then the agent $B_1'$ contains the object $\boxed{ab}$.

For this content of $B_1'$, there exists exactly one applicable program, which replaces the object $\boxed{ab}$ by the object corresponding to the sequence of blindly applicable programs, i.e., by the object $\boxed{\boxed{list}}$, where $list = \varphi(\boxed{ab})$.

| *en.* | $B_1'$ |
|---|---|
| | $\langle \boxed{ab} \to \boxed{\boxed{list}} \rangle$ |
| | $\langle \boxed{\boxed{list}} \leftrightarrow e \rangle$ |
| $\boxed{\boxed{list}}$ | |

The table above has the following structure. In the first column, labelled by *env.*, we list the objects of the environment that are relevant for the given phase of the computation. The object $e$ is not included in this column. Moreover, objects different from $e$ are omitted if the agents listed in the header of the table cannot use them during the considered phase. In the following columns, the programs of the corresponding agents are listed in such an order that each row represents one computation step of the simulation. The current contents of the agents are not explicitly shown, since they can be derived from the left-hand sides of the rules.

Now, agent $B_2'$ has an applicable program. The agent replaces the object $\boxed{\boxed{list}}$ by one of the objects $\boxed{r\,\boxed{list_r}}$, where $list_r$ is the sequence obtained from $list$ by removing the element $r$. The element $r$ is one of the program identifiers contained in $list$. Agent $B_2'$ contains programs that are able to generate $\boxed{r\,\boxed{list_r}}$ for every $r \in list$.

| *en.* | $B_2'$ |
|---|---|
| $\boxed{\boxed{list}}$ | $\langle e \leftrightarrow \boxed{\boxed{list}} \rangle$ |
| | $\langle \boxed{\boxed{list}} \to \boxed{r\,\boxed{list_r}} \rangle$ if list contains at least two members or |
| | $\langle \boxed{\boxed{list}} \to \boxed{r\,\boxed{\emptyset}} \rangle$ if list contains one member or |
| | $\langle \boxed{r\,\boxed{list}} \to \boxed{\boxed{\emptyset}\,\boxed{\emptyset}} \rangle$ if list was already empty |
| | $\langle \boxed{r\,\boxed{list_r}} \leftrightarrow e \rangle$ |
| $\boxed{r\,\boxed{list_r}}$ | |
| | $\langle \boxed{r\,\boxed{\emptyset}} \leftrightarrow e \rangle$ |
| | $\langle \boxed{\boxed{\emptyset}\,\boxed{\emptyset}} \leftrightarrow e \rangle$ |

During the continuation of the simulation, we encounter a problem that has to be solved by every P colony of capacity 1. If an agent generates an object representing some information, this information cannot be propagated through the whole P colony, because only one copy of the object exists in the colony. If an agent sends this object to the environment, it has no possibility to remember the performed action. Moreover, if another agent removes the object from the environment, the information is available only to this agent.

In order to verify whether the selected blindly applicable program is actually applicable, we have to check the presence of up to two objects in the environment, generate up to two new objects into the environment, and create one object representing the new content of the simulated agent. To perform these operations, we introduce additional objects representing information about the currently simulated program and the current state of the simulation.

The generation of these auxiliary objects is performed in pairs. Starting from one object, we generate two objects that are either identical or different, depending on the required simulation step.

The phase responsible for generating two copies of an information object is carried out by four agents, denoted by $B'_{d1}, \ldots, B'_{d4}$. The first agent becomes active as soon as an object representing the corresponding type of information appears in the environment. Let this object be $W_r$, where $r$ is the identifier of the currently simulated program.

Agent $B'_{d1}$ initiates the duplication process by placing the objects $H_{d1}$ and $W'_r$ into the environment.

The agents $B'_{d3}$ and $B'_{d4}$ are responsible for generating auxiliary objects $\boxed{1}, \ldots, \boxed{h}$. The generation is performed sequentially by rewriting one auxiliary object into the next one and transferring the generated objects through the environment. The agent $B'_{d3}$ generates the objects with odd numbers, while the agent $B'_{d4}$ generates the objects with even numbers.

The activity of agent $B'_{d2}$ is triggered by the presence of the object $W'_r$ in the environment. Agent $B'_{d2}$ removes the object $\boxed{r}$ from the environment, thereby interrupting the generation process of the auxiliary objects. Subsequently, the terminating phase of the duplication is initiated, during which two identical objects are generated.

| | *env.* | $B'_{d1}$ | $B'_{d2}$ | $B'_{d3}$ | $B'_{d4}$ |
|---|---|---|---|---|---|
| | | $\langle e \to H_{d1} \rangle$ | | $\langle e \to d_{d3} \rangle$ | $\langle a \to d'_{d3} \rangle$ |
| 1. | $W_r$ | $\langle H_{d1} \leftrightarrow W_r \rangle$ | | | |
| 2. | $H_{d1}$ | $\langle W_r \to W'_r \rangle$ | | $\langle d_{d3} \leftrightarrow H_{d1} \rangle$ | |
| 3. | $d_{d3}$ | $\langle W'_r \leftrightarrow e \rangle$ | | $\langle H_{d1} \to H'_{d1} \rangle$ | $\langle d'_{d3} \leftrightarrow d_{d3} \rangle$ |
| 4. | $W'_r$ $d'_{d3}$ | $\langle e \to H_{d1} \rangle$ | $\langle e \leftrightarrow W'_r \rangle$ | $\langle H'_{d1} \to \boxed{1} \rangle$ | $\langle d_{d3} \to \overline{d} \rangle$ |
| 5. | | | $\langle W'_r \to r'_{d3} \rangle$ | $\langle \boxed{1} \leftrightarrow d'_{d3} \rangle$ | $\langle \overline{d} \to \overline{d'} \rangle$ |
| 6. | $\boxed{1}$ | | | $\langle d'_{d3} \to d'''_{d3} \rangle$ | $\langle \overline{d'} \to d''_{d3} \rangle$ |
| 7. | $\boxed{1}$ | | | | $\langle d''_{d3} \leftrightarrow \boxed{1} \rangle$ |
| 8. | $d''_{d3}$ | | | $\langle d'''_{d3} \leftrightarrow d''_{d3} \rangle$ | $\langle \boxed{1} \to \boxed{2} \rangle$ |
| 9. | $d'''_{d3}$ | | | $\langle d''_{d3} \to \overline{d} \rangle$ | $\langle \boxed{2} \leftrightarrow d'''_{d3} \rangle$ |
| 10. | $\boxed{2}$ | | | $\langle \overline{d} \to d^{iv}_{d3} \rangle$ | $\langle d'''_{d3} \to d^{v}_{d3} \rangle$ |
| 11. | $\boxed{2}$ | | | $\langle d^{iv}_{d3} \leftrightarrow \boxed{2} \rangle$ | |
| 12. | $d^{iv}_{d3}$ | | | $\langle \boxed{2} \to \boxed{3} \rangle$ | $\langle d^{v}_{d3} \leftrightarrow d^{iv}_{d3} \rangle$ |
| 13. | $d^{v}_{d3}$ | | | $\langle \boxed{3} \leftrightarrow d^{v}_{d3} \rangle$ | $\langle d^{iv}_{d3} \to \overline{d_4} \rangle$ |
| 14. | $\boxed{3}$ | | | $\langle d^{v}_{d3} \to d'''_{d3} \rangle$ | $\langle \overline{d_4} \to d''_{d3} \rangle$ |
| | $\vdots$ | | | | |

For agents $B'_{d3}$ and $B'_{d4}$, the following steps (and the execution of the corresponding programs) are repeated with increasing numbers in the framed objects.

| | *env.* | $B'_{d3}$ | $B'_{d4}$ |
|---|---|---|---|
| $m$. | $d^{iv}_{d3}$ | $\left\langle \boxed{i}' \to \boxed{i{+}1} \right\rangle$ | $\langle d^{v}_{d3} \leftrightarrow d^{iv}_{d3} \rangle$ |
| $(m+1)$. | $d^{v}_{d3}$ | $\left\langle \boxed{i{+}1} \leftrightarrow d^{v}_{d3} \right\rangle$ | $\langle d^{iv}_{d3} \to \overline{d_4} \rangle$ |
| $(m+2)$. | $\boxed{i{+}1}$ | $\langle d^{v}_{d3} \to d'''_{d3} \rangle$ | $\langle \overline{d_4} \to d''_{d3} \rangle$ |
| $(m+3)$. | $\boxed{i{+}1}$ | | $\left\langle d''_{d3} \leftrightarrow \boxed{i{+}1} \right\rangle$ |
| $(m+4)$. | $d''_{d3}$ | $\langle d'''_{d3} \leftrightarrow d'' \rangle$ | $\left\langle \boxed{i{+}1} \to \boxed{i{+}2} \right\rangle$ |
| $(m+5)$. | $d'''_{d3}$ | $\langle d''_{d3} \to \overline{d} \rangle$ | $\left\langle \boxed{i{+}2} \leftrightarrow d'''_{d3} \right\rangle$ |
| $(m+6)$. | $\boxed{i{+}2}$ | $\langle \overline{d} \to d^{iv}_{d3} \rangle$ | $\langle d'''_{d3} \to d^{v}_{d3} \rangle$ |
| $(m+7)$ | $\boxed{i{+}2}$ | $\left\langle d^{iv}_{d3} \leftrightarrow \boxed{i{+}2} \right\rangle$ | |

In the following tables, we present the programs of the auxiliary agents and specify the conditions under which they are applied, namely when the environment contains a framed object whose index corresponds to the object that has to be generated in the target copy.

We now describe the terminating phase of the duplication process. This phase involves the auxiliary agents $B'_{d2}$, $B'_{d3}$, and $B'_{d4}$.

Let $r$ be the identifier of the selected program of the original P colony $\Pi$. The purpose of this phase is to terminate the generation of framed objects when the object $\boxed{r}$ is reached and to produce the required copies corresponding to the selected program. Since the generation of framed objects is alternately performed by agents $B'_{d3}$ and $B'_{d4}$, the programs of these agents depend on the parity of $r$. Therefore, the construction distinguishes between the cases when $r$ is odd and when $r$ is even.

Odd numbers

| | *env.* | $B'_{d2}$ | $B'_{d3}$ | $B'_{d4}$ |
|---|---|---|---|---|
| 1. | $\boxed{r}$ | $\langle r'_{d3} \leftrightarrow \boxed{r} \rangle$ | $\langle d^{v}_{d3} \to d'''_{d3} \rangle$ | $\langle \overline{d_4} \to d''_{d3} \rangle$ |
| 2. | $r'_{d3}$ | $\langle \boxed{r} \to r''_{d3} \rangle$ | $\langle d'''_{d3} \leftrightarrow r'_{d3} \rangle$ | |
| 3. | $d'''_{d3}$ | $\langle r''_{d3} \leftrightarrow d'''_{d3} \rangle$ | $\langle r'_{d3} \to r_1 \rangle$ | |
| 4. | $r''_{d3}$ | $\langle d'''_{d3} \to d_{d3} \rangle$ | $\langle r_1 \to r_2 \rangle$ | $\langle d''_{d3} \leftrightarrow r''_{d3} \rangle$ |
| 5. | $d''_{d3}$ | $\langle d_{d3} \leftrightarrow d''_{d3} \rangle$ | $\langle r_2 \to r_3 \rangle$ | $\langle r''_{d3} \to r_1 \rangle$ |
| 6. | $d_{d3}$ | $\langle d''_{d3} \to d'_{d3} \rangle$ | $\langle r_3 \to r_4 \rangle$ | $\langle r_1 \to r_2 \rangle$ |
| 7. | $d_{d3}$ | $\langle d'_{d3} \leftrightarrow e \rangle$ | $\langle r_4 \to \textcircled{r} \rangle$ | $\langle r_2 \to \textcircled{r} \rangle$ |
| 8. | $d_{d3}d'_{d3}$ | | $\langle \textcircled{r} \leftrightarrow d_{d3} \rangle$ | $\langle \boxed{r} \leftrightarrow d'_{d3} \rangle$ |

Even numbers

| | *env.* | $B'_{d2}$ | $B'_{d3}$ | $B'_{d4}$ |
|---|---|---|---|---|
| 1. | $\boxed{r}$ | $\langle r'_{d3} \leftrightarrow \boxed{r} \rangle$ | $\langle \overline{d} \to d^{iv}_{d3} \rangle$ | $\langle d'''_{d3} \to d^{v}_{d3} \rangle$ |
| 2. | $r'_{d3}$ | $\langle \boxed{r} \to r''_{d3} \rangle$ | | $\langle d^{v}_{d3} \leftrightarrow r'_{d3} \rangle$ |
| 3. | $d^{v}_{d3}$ | $\langle r''_{d3} \leftrightarrow d^{v}_{d3} \rangle$ | | $\langle r'_{d3} \to r_1 \rangle$ |
| 4. | $r''_{d3}$ | $\langle d^{v}_{d3} \to d_{d3} \rangle$ | $\langle d^{iv}_{d3} \leftrightarrow r''_{d3} \rangle$ | $\langle r_1 \leftrightarrow r_2 \rangle$ |
| 5. | $d^{iv}_{d3}$ | $\langle d_{d3} \leftrightarrow d^{iv}_{d3} \rangle$ | $\langle r''_{d3} \to r_1 \rangle$ | $\langle r_2 \to r_3 \rangle$ |
| 6. | $d_{d3}$ | $\langle d^{iv}_{d3} \to d'_{d3} \rangle$ | $\langle r_1 \to r_2 \rangle$ | $\langle r_3 \to r_4 \rangle$ |
| 7. | $d_{d3}$ | $\langle d'_{d3} \leftrightarrow e \rangle$ | $\langle r_2 \to \textcircled{r} \rangle$ | $\langle r_4 \to \textcircled{r} \rangle$ |
| 8. | $d_{d3}d'_{d3}$ | | $\langle \textcircled{r} \leftrightarrow d_{d3} \rangle$ | $\langle \textcircled{r} \leftrightarrow d'_{d3} \rangle$ |

The indices $d1$ and $d3$ identify agents belonging to the same group of four agents responsible for duplicating information objects. These indices are attached to all auxiliary objects that may appear in the environment during this duplication process. Consequently, only the intended duplication group can process these objects, preventing the activation of any other duplication group.

We are now ready to continue with the description of the simulation of one computation step of the original P colony $\Pi$.

Since the capacity of the original P colony $\Pi$ is equal to 2, every program consists of exactly two rules. Consequently, there are three possible types of programs: $\langle a \to b;\ c \to g \rangle$, $\langle a \leftrightarrow b;\ c \to g \rangle$, and $\langle a \leftrightarrow b;\ c \leftrightarrow g \rangle$.

Assume that the object $\boxed{r \mid list_r}$ has just appeared in the environment. The identifier $r$ contained in this object corresponds to one of the three program types listed above. We consider these cases separately.

First, let $r$ be the program $\langle a \leftrightarrow b;\ c \leftrightarrow g \rangle$ and let $list = \varphi(\boxed{ac}) = (r_1, \ldots, r_k)$. Then $r$ is one of the elements of the sequence $list$, and $list_r$ is obtained from $list$ by removing the identifier $r$. The object $\boxed{r \mid list_r}$ therefore represents the selected program together with the identifiers of all remaining blindly applicable programs.

The object $\boxed{r \mid list_r}$ placed in the environment initiates the first duplication phase. It serves as the object $W_r$ for the first group of duplication agents. This group transforms the object $W_r = \boxed{r \mid list_r}$ into the objects $\boxed{r \mid list_r}''$ and $D_r$.

For the programs of the duplication agents, the parameters are instantiated as follows. In the tables describing the duplication agents, we substitute $r' = \boxed{r \mid list_r}'''$. In the programs of agent $B'_{d3}$, we set $r_i = \boxed{r \mid list_r}_i$ for $1 \leq i \leq 4$ and

$\textcircled{r} = \boxed{r}\boxed{list_r}''$. In the programs of agent $B'_{d4}$, we substitute $\textcircled{r} = D_r$. At the end of this duplication phase, the objects $\boxed{r}\boxed{list_r}''$ and $D_r$ are present in the environment.

The object $D_r$ activates the second group of duplication agents by setting $W_r = D_r$ and $\textcircled{r} = C_r$. After this duplication phase, two copies of the object $C_r$ are present in the environment. Each of them activates another duplication group with $W_r = C_r$. As a result of this third duplication phase, four copies of the object $\textcircled{r}$ are produced. Notice that all four copies appear in the system simultaneously.

These four copies are used as follows. Two copies are employed to verify the presence of the objects $b$ and $g$ in the environment, while the remaining two copies are used to verify that these objects have been successfully removed from the environment.

Both of these operations require three agents in each group. We can assume that these agents are $B'_3, \ldots, B'_8$.

The first verification is performed by agents $B'_3$ and $B'_4$.

| | *env.* | $B'_3$ | $B'_4$ | $B'_5$ |
|---|---|---|---|---|
| 1. | $\textcircled{r}\textcircled{r}bg$ | $\langle e \leftrightarrow \textcircled{r} \rangle$ | $\langle e \leftrightarrow \textcircled{r} \rangle$ | |
| 2. | $bg$ | $\langle \textcircled{r} \rightarrow r_b \rangle$ | $\langle \textcircled{r} \rightarrow \overline{r_g} \rangle$ | |
| 3. | $bg$ | $\langle r_b \leftrightarrow b \rangle$ | $\langle \overline{r_g} \leftrightarrow g \rangle$ | |
| 4. | $r_b\overline{r_g}$ | $\langle b \rightarrow e \rangle$ | $\langle g \rightarrow e \rangle$ | |

If the object $b$ or $g$ is not present in the environment, the corresponding agent does not place the object $r_b$ or $\overline{r_g}$ into the environment and remains blocked until it is released by an object generated in the environment during the subsequent verification phase.

This subsequent verification phase is performed by agents $B'_6$, $B'_7$, and $B'_8$.

| | *env.* | $B'_6$ | $B'_7$ | $B'_8$ |
|---|---|---|---|---|
| 1. | $\textcircled{r}\textcircled{r}$ | $\langle e \leftrightarrow \textcircled{r} \rangle$ | $\langle e \leftrightarrow \textcircled{r} \rangle$ | |
| 2. | | $\langle \textcircled{r} \rightarrow \textcircled{r'} \rangle$ | $\langle \textcircled{r} \rightarrow \textcircled{r'} \rangle$ | |
| 3. | | $\langle \textcircled{r'} \rightarrow r'' \rangle$ | $\langle \textcircled{r'} \rightarrow r'_1 \rangle$ | |
| **4.** | $r_b\overline{r_g}$ | $\langle r'' \leftrightarrow r_b \rangle$ | $\langle r'_1 \rightarrow r'_2 \rangle$ | |
| 5. | $r''\overline{r_g}$ | $\langle r_b \rightarrow r_0 \rangle$ | $\langle r'_2 \leftrightarrow e \rangle$ | $\langle e \leftrightarrow r'' \rangle$ |
| **6.** | $\overline{r_g}r'_2$ | $\langle r_0 \leftrightarrow \overline{r_g} \rangle$ | | $\langle r'' \rightarrow r_3 \rangle$ |
| 7. | $r_0r'_2$ | $\langle \overline{r_g} \rightarrow r'_{ok} \rangle$ | $\langle e \leftrightarrow r_0 \rangle$ | $\langle r_3 \leftrightarrow e \rangle$ |
| 8. | $r_3r'_2$ | $\langle r'_{ok} \leftrightarrow e \rangle$ | $\langle r_0 \rightarrow d \rangle$ | |
| 9. | $r_3r'_2r'_{ok}$ | | $\langle d \leftrightarrow r_3 \rangle$ | $\langle e \leftrightarrow r'_{ok} \rangle$ |
| 10. | $dr'_2$ | | $\langle r_3 \rightarrow e \rangle$ | $\langle r'_{ok} \rightarrow D_{r_{ok}} \rangle$ |
| 11. | $dr'_2$ | | | $\langle D_{r_{ok}} \leftrightarrow r'_2 \rangle$ |
| 12. | $dD_{r_{ok}}$ | | | $\langle r'_2 \rightarrow e \rangle$ |

If the object $r_b$ is not present in the environment, the computation proceeds differently from step 4. onwards.

| | *env.* | $B'_6$ | $B'_7$ | $B'_8$ |
|---|---|---|---|---|
| **4.** | $\overline{r_g}$ | | $\langle r'_1 \to r'_2 \rangle$ | |
| 5. | $\overline{r_g}$ | | $\langle r'_2 \leftrightarrow e \rangle$ | |
| 6. | $\overline{r_g} r'_2$ | $\langle r'' \leftrightarrow r'_2 \rangle$ | | |
| 7. | $\overline{r_g} r''$ | $\langle r'_2 \to r'_{ko1} \rangle$ | | $\langle e \leftrightarrow r'' \rangle$ |
| 8. | $\overline{r_g}$ | $\langle r'_{ko1} \leftrightarrow e \rangle$ | $\langle r'' \to r_3 \rangle$ | |
| 9. | $\overline{r_g} r'_{ko1}$ | | $\langle e \leftrightarrow r'_{ko1} \rangle$ | $\langle r_3 \leftrightarrow e \rangle$ |
| 10. | $\overline{r_g} r_3$ | | $\langle r'_{ko1} \to D_{r_{ko1}} \rangle$ | |
| 11. | $\overline{r_g} r_3$ | | $\langle D_{r_{ko1}} \leftrightarrow r_3 \rangle$ | |
| 12. | $\overline{r_g} D_{r_{ko1}}$ | | $\langle r_3 \to e \rangle$ | |

If the object $r_b$ was present in the environment, but the object $\overline{r_g}$ was not present, then the computation proceeds from step 6. as follows:

| | *env.* | $B'_6$ | $B'_7$ | $B'_8$ |
|---|---|---|---|---|
| **6.** | $r'_2$ | | | $\langle r'' \to r_3 \rangle$ |
| 7. | $r'_2$ | | | $\langle r_3 \leftrightarrow e \rangle$ |
| 8. | $r_3 r'_2$ | $\langle r_0 \leftrightarrow r_3 \rangle$ | | |
| 9. | $r_0 r'_2$ | $\langle r_3 \to r'_{ko2} \rangle$ | $\langle e \leftrightarrow r_0 \rangle$ | |
| 10. | $r'_2$ | $\langle r'_{ko2} \leftrightarrow e \rangle$ | $\langle r_0 \to d \rangle$ | |
| 11. | $r'_2 r'_{ko2}$ | | $\langle d \leftrightarrow r'_{ko2} \rangle$ | |
| 12. | $dr'_2$ | | $\langle r'_{ko2} \to r''_{ko2} \rangle$ | |
| 13. | $dr'_2$ | | $\langle r''_{ko2} \leftrightarrow e \rangle$ | |
| 14. | $dr'_2 r''_{ok}$ | | | $\langle e \leftrightarrow r''_{ko2} \rangle$ |
| 15. | $dr'_2$ | | | $\langle r''_{ko2} \to D_{r_{ko2}} \rangle$ |
| 16. | $dr'_2$ | | | $\langle D_{r_{ko2}} \leftrightarrow r'_2 \rangle$ |
| 17. | $dD_{r_{ko2}}$ | | | $\langle r'_2 \to e \rangle$ |

At this point of the computation, the environment contains, among other objects, an object whose identifier starts with $D$. Another duplication phase is therefore initiated.

For the objects $D_{r_{ko1}}$ and $D_{r_{ko2}}$, the duplication is performed only once, producing two copies of the corresponding objects $(r_{ko1})$ or $(r_{ko2})$, respectively.

For the object $D_{r_{ok}}$, the information duplication is performed twice. In the first duplication phase, we set $W_r = D_{r_{ok}}$ and $(r) = D_{r'_{ok}}$. In the second duplication phase, we set $W_r = D_{r'_{ok}}$ and $(r) = (r_{ok})$.

Let us first consider the case of a successful verification. In this case, four copies of the object $(r_{ok})$ are present in the environment. Two of these copies are used by agents $B'_9$ and $B'_{10}$ to generate new objects in the environment. These objects correspond to the objects inserted into the environment by the original agent of the P colony $\Pi$, namely the objects $a$ and $c$.

| | *env.* | $B'_9$ | $B'_4$ | $B'_{10}$ |
|---|---|---|---|---|
| 1. | $(r_{ok})(r_{ok})$ | $\langle e \leftrightarrow (r_{ok}) \rangle$ | $\langle e \leftrightarrow (r_{ok}) \rangle$ | |
| 2. | | $\langle (r_{ok}) \to a \rangle$ | $\langle (r_{ok}) \to c \rangle$ | |
| 3. | | $\langle a \leftrightarrow e \rangle$ | $\langle c \leftrightarrow e \rangle$ | |
| 4. | $ac$ | | | |

While the verification of the environment contents was being performed, the object $[\,r\,|\,list\,]''$ was taken by agent $B'_{11}$ and rewritten into $[\,r\,|\,list\,]^{ok}$. After this step, the agent cyclically rewrites $[\,r\,|\,list\,]^{ok}$ into $[\,r\,|\,list\,]^{ko}$ and vice versa.

The nondeterministic termination of this cyclic rewriting occurs when one of the programs replacing $[\,r\,|\,list\,]^{ok}$ by $r_{ok}$ or $[\,r\,|\,list\,]^{ko}$ by $r_{ko3}$ is applied. The latter object $r_{ko3}$ is generated after an unsuccessful verification and after the release of agents $B'_3$ and $B'_4$ from their blocked state.

| | $en.$ | $B'_{11}$ |
|---|---|---|
| 1. | $[\,r\,\|\,list\,]''$ | $\langle e \leftrightarrow [\,r\,\|\,list\,]'' \rangle$ |
| 2. | | $\langle [\,r\,\|\,list\,]'' \rightarrow [\,r\,\|\,list\,]^{ok} \rangle$ |
| 3. | | $\langle [\,r\,\|\,list\,]^{ok} \rightarrow [\,r\,\|\,list\,]^{ko} \rangle$ |
| 4. | | $\langle [\,r\,\|\,list\,]^{ko} \rightarrow [\,r\,\|\,list\,]^{ok} \rangle$ |
| $\vdots$ | | |
| $m.$ | | $\langle [\,r\,\|\,list\,]^{ok} \leftrightarrow (r_{ok}) \rangle$ |
| $m'.$ | | $\langle [\,r\,\|\,list\,]^{ko} \leftrightarrow (r_{ko3}) \rangle$ |

The objects $[\,r\,|\,list\,]^{ok}$ and $[\,r\,|\,list\,]^{ko}$ are consumed by agent $B'_2$, which processes them in the next phase of the simulation.

| $en.$ | $B'_2$ |
|---|---|
| $[\,r\,\|\,list\,]^{ok}$ | $\langle e \leftrightarrow [\,r\,\|\,list\,]^{ok} \rangle$ |
| | $\langle [\,r\,\|\,list\,]^{ok} \rightarrow e \rangle$ |
| $[\,r\,\|\,list\,]^{ko}$ | $\langle e \leftrightarrow [\,r\,\|\,list\,]^{ko} \rangle$ |
| | $\langle [\,r\,\|\,list\,]^{ko} \rightarrow [\,\|\,list\,] \rangle$ |

The third copy of the object $(r_{ok})$ is used by agent $B'_1$, which generates its new content from this object.

| $en.$ | $B'_1$ |
|---|---|
| $(r_{ok})$ | $\langle e \leftrightarrow (r_{ok}) \rangle$ |
| | $\langle (r_{ok}) \rightarrow [\,bg\,] \rangle$ |

The last part of the simulation that remains to be described concerns the situation when the verification reveals that one of the objects required for the execution of the program $r$ is missing in the environment. This situation is detected by the appearance of $(r_{ko1})$ or $(r_{ko2})$ in the environment.

If the object $(r_{ko1})$ appears, it means that the object $b$ was not present in the environment. There are two possible cases: either the object $g$ was present or it was not present (agent $B'_4$ is either active or blocked, respectively). If neither of the two objects was present in the environment, the following programs are applied:

| | *env.* | $B'_3$ | $B'_4$ | $B'_5$ |
|---|---|---|---|---|
| 1. | $\textcircled{r_{ko1}}\textcircled{r_{ko1}}$ | $\langle r_b \leftrightarrow \textcircled{r} \rangle$ | $\langle \overline{r_g} \leftrightarrow \textcircled{r_{ko1}} \rangle$ | |
| 2. | $r_b\overline{r_g}$ | $\langle \textcircled{r_{ko1}} \rightarrow r'_{ko1} \rangle$ | $\langle \textcircled{r_{ko1}} \rightarrow r''_{ko1} \rangle$ | |
| 3. | | $\langle r'_{ko1} \leftrightarrow \overline{r_g} \rangle$ | $\langle r''_{ko1} \leftrightarrow r_b \rangle$ | |
| 4. | $r'_{ko1}r''_{ko1}$ | $\langle \overline{r_g} \rightarrow e \rangle$ | $\langle r_b \rightarrow e \rangle$ | $\langle e \leftrightarrow r'_{ko1} \rangle$ |
| 5. | $r''_{ko1}$ | | | $\langle r'_{ko1} \rightarrow r_{ko3} \rangle$ |
| 6. | $r''_{ko1}$ | | | $\langle r_{ko3} \leftrightarrow r''_{ko1} \rangle$ |
| 7. | $r_{ko3}$ | | | $\langle r''_{ko1} \rightarrow e \rangle$ |

If the object $b$ was not present in the environment, but the object $g$ was present, the computation proceeds as follows:

| | *env.* | $B'_3$ | $B'_4$ $B'_5$ |
|---|---|---|---|
| 1. | $\textcircled{r_{ko1}}\textcircled{r_{ko1}}\overline{r_g}$ | $\langle r_b \leftrightarrow \textcircled{r} \rangle$ | |
| 2. | $\textcircled{r_{ko1}}r_b\overline{r_g}$ | $\langle \textcircled{r_{ko1}} \rightarrow r'_{ko1} \rangle$ | |
| 3. | $\textcircled{r_{ko1}}r_b\overline{r_g}$ | $\langle r'_{ko1} \leftrightarrow \overline{r_g} \rangle$ | |
| 4. | $\textcircled{r_{ko1}}r'_{ko1}r_b$ | $\langle \overline{r_g} \rightarrow e \rangle$ | $\langle e \leftrightarrow r'_{ko1} \rangle$ |
| 5. | $\textcircled{r_{ko1}}r_b$ | | $\langle r'_{ko1} \rightarrow r_{ko3} \rangle$ |
| 6. | $\textcircled{r_{ko1}}r_b$ | | $\langle r_{ko3} \leftrightarrow \textcircled{r_{ko1}} \rangle$ |
| 7. | $r_{ko3}r_b$ | | $\langle \textcircled{r_{ko1}} \rightarrow d \rangle$ |
| 8. | $r_{ko3}r_b$ | | $\langle d \leftrightarrow r_b \rangle$ |
| 9. | $r_{ko3}d$ | | $\langle r_b \rightarrow e \rangle$ |

The last case to consider is when the object $b$ was present in the environment, while the object $g$ was not present. We will need agent $B'_{12}$.

| | *env.* | $B'_3$ $B'_4$ | $B'_{12}$ |
|---|---|---|---|
| 1. | $\textcircled{r_{ko2}}\textcircled{r_{ko2}}$ | $\langle \overline{r_g} \leftrightarrow \textcircled{r_{ko2}} \rangle$ | $\langle e \leftrightarrow \textcircled{r_{ko2}} \rangle$ |
| 2. | $\overline{r_g}$ | $\langle \textcircled{r_{ko2}} \rightarrow r''_{ko2} \rangle$ | $\langle \textcircled{r_{ko2}} \rightarrow g \rangle$ |
| 3. | $\overline{r_g}$ | $\langle r''_{ko2} \leftrightarrow e \rangle$ | $\langle g \leftrightarrow r_g \rangle$ |
| 4. | $r''_{ko2}$ | | $\langle r_g \rightarrow d \rangle$ |
| 5. | $r''_{ko2}$ | | $\langle d \leftrightarrow r''_{ko2} \rangle$ |
| 6. | $d$ | | $\langle r''_{ko2} \rightarrow r_{ko3} \rangle$ |
| 7. | $d$ | | $\langle r_{ko3} \leftrightarrow e \rangle$ |

For a program of the form $r : \langle a \leftrightarrow b;\ c \rightarrow g \rangle$, the verification proceeds similarly. Agent $B'_3$ checks the presence of the object $b$ in the environment, while agent $B'_4$ simply rewrites the object $\textcircled{r}$ into $e$. The subsequent verification phase, performed by agents $B'_6$, $B'_7$, and $B'_8$, only detects the presence of the object $r_b$ in the environment.

If the verification fails, only agent $B'_3$ has to be released from its blocked state; no objects need to be restored to the environment. Immediately after the release of agent $B'_3$, the object $r_{ko3}$ is generated, and the remainder of the simulation proceeds in the same way as in the previous case.

The agents responsible for modifying the environment behave either exactly as in the previous case (for the object $a$), or simply rewrite the object $\textcircled{r_{ok}}$ into $e$.

For a program of the form $r : \langle a \to b;\ c \to g \rangle$, no verification is required. The objects $(r)$ are consumed by the corresponding agents, which rewrite them into $e$, except for agent $B_6'$, which rewrites its copy into $D_{r_{ok}}$. The subsequent modification of the environment consists only of consuming the objects $(r_{ok})$ and rewriting them into $e$. The behaviour of agent $B_2'$ remains unchanged.

The complete specification of the remaining agents and their programs follows the same principles as those presented above. Since their construction is straightforward and differs only in the particular objects being processed, we leave the full listing of these programs to the reader.

The constructed P colony consists of 43 agents. Their roles are as follows:

- one agent simulating the original agent,
- one agent selecting a blindly applicable program and generating the remaining program list,
- 32 agents forming eight groups responsible for information duplication,
- three agents performing the verification of communication objects,
- three agents verifying the result of the communication check,
- two agents generating new objects in the environment,
- one agent responsible for adding the index `ok` or `ko` to the object representing the currently simulated program together with the corresponding list of alternative programs,

Hence, the total number of agents is 43.

The construction described above simulates every computation step of the original one-agent P colony by a finite sequence of computation steps of the constructed P colony. Every applicable program of the original agent is simulated correctly, including all possible outcomes of the communication tests, and the content of the simulated agent as well as the environment are updated exactly as in the original system. If no program is blindly applicable for the current content of the simulated agent, the object $\boxed{\emptyset}\boxed{\emptyset}$ is generated. Its appearance indicates that the original agent has no applicable program in the current configuration, and therefore the computation of the constructed P colony also halts. Conversely, every successful simulation in the constructed P colony corresponds to the application of one program of the original P colony. Hence, the two P colonies perform equivalent computations and generate the same language. Therefore, the theorem holds.

□

**Acknowledgments.** This work was supported by Silesian University in Opava within the project IGS/1/2026 and SGS/9/2024.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Cienciala, L., Ciencialová, L.: P colonies and their extensions. In: Kelemen, J., Kelemenová, A. (eds.) Computation, Cooperation, and Life, pp. 158–169. Springer-Verlag, Berlin, Heidelberg (2011)

2. Cienciala, L., Ciencialová, L., Kelemenová, A.: On the number of agents in P colonies. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **4860 LNCS**, 193–208 (2007)
3. Ciencialová, L., Csuhaj-Varjú, E., Kelemenová, A., Vaszil, Gy.: Variants of P colonies with very simple cell structure. International Journal of Computers, Communications and Control **4**(3), 224–233 (2009)
4. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, Gy.: Computing with cells in environment: P colonies. Journal of Multiple-Valued Logic and Soft Computing **12**(3-4 SPEC. ISS.), 201–215 (2006)
5. Csuhaj-Varjú, E., Margenstern, M., Vaszil, Gy.: P colonies with a bounded number of cells and programs. In: Hoogeboom, H.J., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers. Lecture Notes in Computer Science, vol. 4361, pp. 352–366. Springer (2006). https://doi.org/10.1007/11963516_22
6. Freund, R., Oswald, M.: P colonies working in the maximally parallel and in the sequential mode. In: Zaharie, D., Petcu, D., Negru, V., Jebelean, T., Ciobanu, G., Cicortas, A., Abraham, A., Paprzycki, M. (eds.) Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005), 25-29 September 2005, Timisoara, Romania. pp. 419–426. IEEE Computer Society (2005). https://doi.org/10.1109/SYNASC.2005.55
7. Kelemen, J., Kelemenová, A.: On P colonies, a biochemically inspired model of computation. In: Proc. of the $6^{th}$ International Symposium of Hungarian Researchers on Computational Intelligence, Budapest TECH. pp. 40–56. Hungary (2005), http://conf.uni-obuda.hu/mtn2005/Kelemen.pdf
8. Kelemen, J., Kelemenová, A., Păun, Gh.: Preview of P colonies: A biochemically inspired computing model. In: Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX). pp. 82–86. Boston, Massachusetts, USA (September 12-15 2004)

# Scalable Verification-Aware Kernel P Systems: Indexed Modelling, Candidate Generation, and Trace-Level Verification

Doru Constantin[3], Cristina Tudose[3], Ștefan Iacob[1], Ionuț Mihai Niculescu[3], Florentin Ipate[1], and Marian Gheorghe[2]

[1] University of Bucharest, Faculty of Mathematics and Computer Science, 14 Academiei St., 010014 Bucharest, Romania
f.ipate@fmi.unibuc.ro, stefan.iacob@fmi.unibuc.ro
[2] University of Bradford, School of Computing and Engineering, West Yorkshire, Bradford BD7 1DP, UK
m.gheorghe@bradford.ac.uk
[3] National University of Science and Technology POLITEHNICA Bucharest, Department of Mathematics-Informatics, Pitești University Centre, 110040, Romania
doru.constantin0804@upb.ro, cristina.tudose0203@upb.ro, ionutmihainiculescu@gmail.com

**Abstract.** Kernel P systems provide an executable formalism for parallel rule-based computation with simulation and verification support. We present a verification-aware benchmark methodology combining source-level indexation and refinement, direct kPWorkbench simulation, external candidate generation, execution-derived representative selection, and temporal checking. The methodology is evaluated on Broadcasting, Partition/Subset Sum, and Binary Adder, covering communication, combinatorial branching, and bit-level arithmetic. Refined and indexed sources make completion, pruning, cleanup, and carry propagation more explicit and therefore more directly accessible to property-oriented analysis. At workflow level, generated candidates are materialized and executed in kPWorkbench before representative traces are selected for NuSMV checking. Across the 18 selected representatives, the campaign comprises 1203 real executions and 282 NuSMV properties, all of which hold. A complementary Subset Sum control audit further shows that selected and lower-ranked candidates can be distinguished by non-vacuous target-reachability and positive-verdict properties.

**Keywords:** Membrane computing · Kernel P systems · kP-Lingua · kP-Workbench · Model checking · Indexation · Genetic algorithms

## 1 Introduction

Membrane computing studies computational models inspired by cellular organization and by the evolution and communication of multisets of objects [1–4]. Kernel P systems provide a graph-based membrane-computing formalism

combining typed compartments, object multisets, rewriting and communication rules, guards, and execution strategies [5]. Their executable realization through kP-Lingua and kPWorkbench supports modelling, simulation, and verification-oriented analysis [6]. Recent kP-Lingua preprocessing and indexation mechanisms further allow parameterized families of rules, objects, guards, and links to be expressed compactly [7].

Current membrane-computing research continues to develop both the underlying formalisms and their software, application, and optimization perspectives. Recent studies include tool- and model-oriented developments for neural P systems, evolutionary and membrane-inspired optimization strategies, and new formal variants of P systems [31–33]. Related P-Lingua software resources also support the executable-model perspective adopted here [8]. Formal verification provides a complementary foundation: temporal logics and explicit-state or symbolic model checking make it possible to state and evaluate safety, liveness, response, and structural requirements [12–16]. SPIN and NuSMV are used as verification back ends in the experiments reported below [17–19]. A remaining methodological issue is how to connect these capabilities without conflating their roles. A compact indexed specification, a promising descriptor-level candidate, a successful concrete simulation, and a satisfied temporal property constitute different forms of evidence. This paper therefore investigates how source-level indexation and refinement, external candidate generation, real kP-Workbench execution, execution-derived selection, and trace-level verification can be organized as consecutive and reproducible stages. Three complementary benchmarks are used: Broadcasting for communication and cleanup, Partition/-Subset Sum for combinatorial generation and pruning, and Binary Adder for indexed arithmetic and carry propagation. The complete executable sources and the Python/Jupyter implementations of the external layer are distributed with the experimental resources in the public kPWorkbench repository. The paper makes four contributions. First, it defines a benchmark methodology for comparing standard kP sources with indexed or refined specifications while keeping source-level and workflow-level evidence separate. Second, it reports simulation and property-checking results for Broadcasting, Partition/Subset Sum, and Binary Adder. Third, it defines and implements a benchmark-specific external candidate-generation layer in which descriptor-level proxy ranking is followed by source materialization, real execution, and execution-derived representative selection. Fourth, it evaluates selected representatives through finite NuSMV

**Table 1.** Role of the three benchmarks in the comparative study.

| Benchmark | Main modelling issue | Refined/indexed evidence | Experimental focus |
|---|---|---|---|
| Broadcasting | communication under a threshold | direct return rules, stopped markers, and cleanup | threshold reachability and absence of residual messages |
| Partition/Subset Sum | combinatorial branch growth | target-sum pruning and explicit verdict structure | branch reduction and verdict consistency |
| Binary Adder | arithmetic representation | explicit sum bits and carry propagation | canonical output and arithmetic agreement |

trace abstractions and reports both the main verification campaign and a complementary Subset Sum control experiment. The external search follows standard evolutionary-computation principles [25, 26, 28–30], but the formal evidence reported in this study is obtained only from materialized executable sources and their associated verification artefacts.

## 2 Benchmark Problems and Study Scope

Table 1 summarizes the computational role, refined evidence, and experimental focus associated with each benchmark. The study considers three benchmark problems that exercise complementary parts of the kP modelling and verification workflow. The first problem, *Broadcasting*, models the controlled circulation of message families between a central compartment and a set of receivers. It supports the analysis of communication, threshold reachability, post-threshold cleanup, and the distinction between a blocked terminal configuration and a completed execution with explicit stopping evidence. The initial source provides a compact communication baseline, whereas the refined source introduces direct receiver-side return rules and stopped markers. The second problem, *Partition/Subset Sum*, represents a classical combinatorial search based on include/exclude decisions [23, 24]. The initial source provides an exhaustive partition-style baseline, while the pruning-aware source represents target-sum reachability and avoids extending branches that cannot reach a valid target under positive weights. This benchmark supports the analysis of branch growth, pruning, verdict consistency, and source-level safety properties. The third problem, *Binary Adder*, compares arithmetic expressed through object multiplicities with an indexed full-adder representation. The symbolic source preserves numerical value at the multiset level, whereas the indexed source makes input bits, sum bits, and carry objects explicit. It is used to analyse carry propagation, input cleanup, canonical output, and arithmetic agreement. Together, the three benchmarks cover communication behaviour, combinatorial branching, and arithmetic structure within a common comparative workflow. Broadcasting is parameterized by the number of receiver families $k$ and a threshold $\theta$. The central compartment emits message objects $b_i$ as active objects $a_i$, receives acknowledgements, and accumulates a counter object $c$. The main source-level questions are whether the threshold is reached, whether active messages are removed after completion, and whether the final configuration records completion explicitly. For Partition/Subset Sum, let $V = \{v_1, \ldots, v_n\}$, with positive integer weights $w(v_i) = k_i$. Partition asks whether there exist $V_1, V_2$ such that $V_1 \cap V_2 = \emptyset, V_1 \cup V_2 = V, \sum_{v_i \in V_1} w(v_i) = \sum_{v_i \in V_2} w(v_i)$. For a prescribed positive target value $K$, the Subset Sum variant asks whether there exists a subset $V_1 \subseteq V$ such that $\sum_{v_i \in V_1} w(v_i) = K$. Thus, $K$ is part of the problem instance and denotes the target sum against which the generated subsets are evaluated. The initial source exposes the generated subset space, whereas the pruning-aware source uses the target condition to avoid extending branches that cannot reach $K$ under positive weights. For Binary Adder, the symbolic baseline uses multi-

**Table 2.** Main notation.

| Symbol | Meaning | Symbol | Meaning |
|---|---|---|---|
| $\Pi, A, \mu, C_i, i_0$ | kP system components | $d$ | number of compartments |
| $k, \theta$ | Broadcasting size and threshold | $n, K$ | Subset Sum size and target |
| $v_i, w(v_i), k_i$ | item and positive weight | $m_{\text{bit}}$ | Binary Adder bit width |
| $a_i, b_i, c$ | active, returned, and counter objects | $carry_i$ | carry object at position $i$ |
| $F^{\text{proxy}}$ | proxy-fitness score | $V, H$ | validity and reachability indicators |
| $B_{\text{term}}$ | retained terminal branches | $S_{\text{est}}$ | estimated execution stages |

plicity rewriting, $2a_i \to a_{i+1}$, whereas the indexed full-adder source represents each bit position through input, sum, and carry objects. The relevant properties concern both the value represented by the final multiset and the canonical form of the binary output. Throughout the paper, $k$ and $\theta$ denote the Broadcasting size and threshold, $n$ and $K$ denote the Subset Sum size and target, and $m_{\text{bit}}$ denotes the Binary Adder bit width.

## 3 Background, Benchmarks, and Source Variants

Following the standard kernel P-system formalization [5], a kernel P system of degree $d$ is written as $\Pi = (A, \mu, C_1, \ldots, C_d, i_0)$, where $A$ is a finite alphabet of objects, $\mu$ is the graph describing the communication links between compartments, $C_1, \ldots, C_d$ are the compartments of the system, and $i_0$ identifies the output compartment. Each compartment has a type, a current multiset of objects, and an associated rule set together with an execution strategy. Objects represent the local state of a compartment through their multiplicities. Kernel P rules can rewrite objects locally, communicate objects between linked compartments, divide or dissolve compartments, or modify the communication graph. A rule may be guarded by conditions over object multiplicities; such a rule is applicable only when its guard is satisfied in the current configuration. At each computation step, applicable rules are selected according to the strategy associated with the compartment type. Strategies such as *choice* select among enabled alternatives, whereas maximal parallel strategies apply a maximal compatible collection of rules. The simultaneous effects of the selected rules determine the object multisets and, when structural rules are present, the compartment graph of the next configuration. This operational view is the one used by the executable kP-Lingua sources and by the kPWorkbench simulations considered in the experiments. The notation used throughout the source-level and workflow-level analyses is summarized in Table 2. kP-Lingua indexation is used as a syntactic macro mechanism that compactly generates families of rules, objects, guards, and links without changing the underlying operational semantics. Indexed and non-indexed sources are therefore comparable when instantiated with the same parameters. Formal verification is considered at two complementary levels. At source level, KPQ/LTL formulas are checked through the kPWorkbench–SPIN/NuSMV toolchain whenever the required predicates

**Table 3.** Source variants and associated source-level property families.

| Problem | Variants | Main refinement | Property families |
|---|---|---|---|
| Broadcasting | B-NI, B-I, B-R | direct return and stopped markers | threshold, cleanup, residual messages |
| Subset Sum | P-NI, P-I, P-PR | pruning-aware target-sum source | reachability, exclusivity, pruning |
| Binary Adder | BS-NI, BS-I, BA-I | explicit full-adder carry structure | sum bits, carry, cleanup, agreement |

are directly available. At workflow level, selected concrete simulation traces are translated into finite NuSMV abstractions and checked against temporal properties. Safety properties express the absence of undesirable states, liveness properties capture eventual events, and structural invariants describe branch, object, or arithmetic relations. The study uses three benchmarks with complementary computational structures. *Broadcasting* models messages transmitted from a central compartment to receiver compartments under a threshold counter. The refined source introduces direct receiver-side return rules and explicit stopped markers, providing a clear representation of completed execution. *Partition/Subset Sum* starts from exhaustive include/exclude branch generation and then considers a pruning-aware target-sum source in which branches that cannot reach the target under positive weights are not extended. *Binary Adder* starts from symbolic addition based on multiplicity rules $2a_i \to a_{i+1}$ and moves to an indexed full-adder representation with explicit input, sum, and carry objects. These benchmarks jointly cover communication, combinatorial branching, and arithmetic computation. The variant suffixes denote non-indexed (NI), indexed (I), refined (R), and pruning-aware (PR) sources. Accordingly, B, P, BS, and BA denote Broadcasting, Partition/Subset Sum, Binary Symbolic, and Binary Adder sources, respectively. Table 3 identifies the variants considered for each benchmark and their associated source-level property families. The executable source files are available in the public repository cited in the introduction. The following analysis compares their modelling choices, source-level properties, and workflow-level verification evidence.

*Source-Level Modelling Choices.* The source-level comparison focuses on the modelling decisions that determine both execution behaviour and verification readiness. In Broadcasting, the subscript $i$ identifies a message family. The central compartment initially stores an object $b_i$, which becomes the active message $a_i$ when it is sent to a receiver. Receiver $i$ is associated with the local marker $x_i$, used by the initial source to identify the message family that the receiver is expected to acknowledge. Because the target receiver is selected nondeterministically among compartments of the same type, an $a_i$ object can reach a receiver carrying $x_j$, with $j \neq i$; in that configuration, the corresponding guarded acknowledgement rule is not enabled and the active message may remain blocked. The refined source removes this mismatch by using direct receiver-side return rules and introduces explicit `stopped_i` markers to represent completion. The consequences of this modelling change are evaluated later by the source-level simulation and property results, where blocking-induced termination is distinguished from explicit completion.

**Table 4.** Main stages of the external candidate-generation and verification workflow.

| Stage | Role | Evidence retained |
|---|---|---|
| Candidate generation | Build a static set of variants or a population of chromosomes | Candidate descriptors and proxy scores |
| Materialization | Export selected descriptors as executable `.kpl` sources | Concrete source files |
| Real execution | Run generated sources in kPWorkbench | Logs, halting status, steps, objects, verdicts |
| Metric selection | Rank candidates using real metrics | One representative per case |
| Trace checking | Check finite NuSMV abstractions of selected traces | LTL/CTL truth values |

In Partition/Subset Sum, the initial source is retained as an exhaustive branch-generation baseline. It exposes the $2^n$ candidate space and provides a reference for source-level simulation. The pruning-aware source adopts the target-sum interpretation and stops extending branches whose partial sums cannot reach the target under positive weights. The associated checks do not treat the two sources as contradictory; instead, they examine whether directly expressible safety and verdict properties are preserved while the explored branching structure is reduced. In Binary Adder, the symbolic source is value-oriented: multiplicity rules of the form $2a_i \to a_{i+1}$ preserve the numerical value represented by the multiset, although a terminal multiset may not always be canonical. The indexed full-adder source is structure-oriented and represents input bits, sum bits, and carry objects explicitly. This representation may require more execution steps for some instances, but it supports local carry properties, explicit cleanup conditions, and canonical binary output. The comparison is therefore based on arithmetic agreement and verification readiness rather than on step count alone.

## 4 Candidate-Generation and Verification Workflow

The workflow integrates candidate generation, executable simulation, metric-based selection, and temporal verification. Its input consists of a benchmark configuration specifying the source template, case parameters, candidate representation, proxy-fitness components, and execution plan. Its output is a set of case-level representatives supported by real kPWorkbench metrics and temporal-property checks. Table 4 summarizes the main workflow stages and the evidence retained at each stage. The external layer is implemented in Python through the Jupyter notebooks distributed with the public experimental artefacts. Its role and configuration are benchmark-specific. Broadcasting uses a deterministic candidate-generation and proxy-ranking procedure rather than a generational genetic algorithm. Subset Sum and Binary Adder use genetic algorithms operating on descriptor-level chromosomes. In all cases, the external layer precedes kPWorkbench execution: proxy scores guide generation or preliminary ranking, whereas the representatives reported later are selected from metrics extracted from real executable runs. The descriptor-level choices and proxy indicators used for the three benchmarks are summarized in Table 5. For Broadcasting, the deterministic grid contains six cases and six candidate policies per case. The ordering policies are ascending, descending, odd–even, even–odd,

**Table 5.** Descriptor-level choices used by the external layer.

| Benchmark | Candidate descriptor | Proxy-level indicators |
|---|---|---|
| Broadcasting | case parameters and refined source | threshold reachability, cleanup estimate, order smoothness |
| Subset Sum | item ordering and pruning trajectory | structural validity, target reachability, retained branches, estimated stages |
| Binary Adder | block size, carry strategy, and grouping | carry suitability, rule regularity, scalability, verification readiness |

**Table 6.** External candidate-generation configurations used in the experiments.

| Parameter | Broadcasting | Subset Sum | Binary Adder |
|---|---|---|---|
| Mode | deterministic ranking | genetic | genetic |
| Chromosome / descriptor | ordering and stop policy | item-weight permutation | block size, carry strategy, grouping |
| Population size | – | 20 | 30 |
| Generations | – | 20 | 20 |
| Elitism | – | 2 | 3 |
| Selection | – | tournament (3) | tournament (3) |
| Crossover | – | order, $p_c = 0.80$ | uniform gene-wise, $p_c = 0.82$ |
| Mutation | – | swap, $p_m = 0.15$ | single-gene, $p_m = 0.18$ |
| Random seed | deterministic per candidate | 42 | 2026 |
| Export per case | 6 candidates | up to 10 | 6 |

center-out, and deterministically shuffled orderings, combined with benchmark-specific stopping policies. The pseudo-random ordering uses the deterministic seed $2026 + 6(c-1) + j$, where $c$ and $j$ denote the case and candidate indices, respectively. Consequently, regeneration of the same candidate grid does not depend on an uncontrolled random state.

For Subset Sum, a chromosome is a permutation $\chi_{\mathrm{SS}} = \left(w_{\pi(1)}, \ldots, w_{\pi(n)}\right)$ of the item weights. A population of 20 permutations is evolved for 20 generations. At each generation, the two highest-ranked chromosomes are copied unchanged to the next population. The remaining positions are generated from parents selected by tournaments of size three. Order crossover is applied with probability 0.80; otherwise the first selected parent is copied. Swap mutation, applied with probability 0.15, exchanges two randomly selected positions. The run is initialized with random seed 42, and up to the ten highest-ranked descriptors per case are exported for the subsequent executable workflow.

For Binary Adder, a chromosome is $\chi_{\mathrm{BA}} = (\mathit{block\_size}, \mathit{carry\_strategy}, \mathit{grouping})$, where the first gene selects an admissible block size, the second selects one of the serial, block-carry, lookahead-like, or hybrid carry strategies, and the third specifies the ordering/grouping policy used when generating the indexed rule families. A population of 30 chromosomes is evolved for 20 generations. The three highest-ranked chromosomes are retained unchanged. The remaining population is created using tournaments of size three, uniform gene-wise crossover with probability 0.82, and single-gene mutation with probability 0.18. Uniform crossover independently takes each of the three genes from the second parent with probability 0.5; mutation selects one gene and replaces it with another admissible value. The implementation uses random seed 2026 and exports the

**Table 7.** The algorithmic steps of the verification-aware workflow.

| Step | Operation | Scientific role |
|---|---|---|
| 1 | Instantiate benchmark case | fixes $k, \theta$, $n, K$, or $m_{\text{bit}}$ and the source template |
| 2 | Generate candidate descriptor | represents ordering, pruning, grouping, or carry-related design choices |
| 3 | Rank by proxy indicators | filters candidates before real execution |
| 4 | Materialize `.kpl` source | converts descriptors into executable kP-Lingua artefacts |
| 5 | Execute in kPWorkbench | obtains real steps, halting status, objects, verdicts, and rates |
| 6 | Select case representative | uses real metrics for the reported representative |
| 7 | Check NuSMV trace abstraction | evaluates LTL/CTL properties over selected real behaviours |

six highest-ranked candidates per case. The associated execution plan contains ten kPWorkbench runs per exported candidate before real-metric aggregation. Table 7 presents the workflow as a compact sequence of candidate-generation, execution, selection, and verification steps.

The optimization objective of the external layer is to prioritize descriptors that are structurally admissible and operationally promising before the more expensive executable evaluation. The resulting values are therefore *proxy* scores and must not be interpreted as model-checking results. For Subset Sum, the implemented proxy score is $F_{\text{SS}}^{\text{proxy}} = 10000V + 500H - B_{\text{term}} - 0.1S_{\text{est}}$, where $V$ indicates structural validity, $H$ indicates whether the target is reachable in the pruned descriptor-level calculation, $B_{\text{term}}$ is the retained terminal-branch count, and $S_{\text{est}}$ is the estimated number of stages. The coefficients implement a lexicographic preference: structural validity dominates target reachability, while retained branching and estimated depth act as secondary efficiency criteria. For Binary Adder, the implementation combines normalized descriptor scores as $F_{\text{BA}}^{\text{proxy}} = 100\big(0.32C + 0.18B + 0.15G + 0.12R + 0.15S + 0.08Q\big)$, where $C$, $B$, $G$, $R$, $S$, and $Q$ denote carry-strategy suitability, block-size suitability, grouping compatibility, rule regularity, scalability, and verification-readiness indicators, respectively. Their weights express the design-time emphasis of the descriptor ranking and are not learned from the final verification outcomes. After materialization, candidates are evaluated through real kPWorkbench executions. Halting status, execution steps, final objects, residual objects, verdicts, validity, success, cleanup, and benchmark-specific correctness indicators are extracted from the logs. Execution-derived metrics, rather than proxy fitness alone, determine the case representative subsequently submitted to NuSMV. Table 6 fixes the implementation parameters needed to reproduce the candidate-generation stage. In both genetic benchmarks, the replacement mechanism is generational with elitism: the configured number of highest-ranked chromosomes is first copied unchanged, after which tournament selection, probabilistic crossover, and mutation are repeatedly applied until the next population reaches its configured size. The evolutionary process terminates after the fixed 20-generation budget. These operations affect only candidate descriptors; they do not execute kPWorkbench and do not evaluate SPIN or NuSMV properties. Candidate descriptors become verification evidence only after materialization, executable simulation, real-metric extraction, and representative selection. After materialization, each

**Table 8.** Evidence preserved for each benchmark.

| Benchmark | Source-level evidence | Workflow-level evidence | Main interpretation |
|---|---|---|---|
| Broadcasting | SPIN/NuSMV threshold and cleanup checks | 78/78 NuSMV trace checks | explicit completion and cleanup |
| Subset Sum | SPIN safety checks and pruning comparison | 66/66 NuSMV trace checks | reduced branching with preserved verdicts |
| Binary Adder | SPIN/NuSMV arithmetic checks | 138/138 NuSMV trace checks | canonical output through indexed carry structure |

source is evaluated through concrete kPWorkbench executions. The collected metrics include halting status, number of steps, final object multiplicities, residual active objects, threshold or target reachability, completion rate, validity rate, success rate, and, for Binary Adder, arithmetic correctness. Proxy indicators are therefore used for search and preliminary ranking, whereas execution metrics determine the reported case representative. In Broadcasting, the representative is selected according to completion and cleanup behaviour for the given case parameters. In Subset Sum, the selected source preserves the relation between the target state and the reported verdict after pruning. In Binary Adder, selection considers canonical sum bits, carry cleanup, and arithmetic agreement. The subsequent NuSMV checks are applied to these concrete representatives, linking workflow-level verification to executable outputs. The verification layer uses property families adapted to the observable structure of each benchmark. Broadcasting properties address threshold reachability, cleanup after threshold, stopped-family evidence, and the absence of residual active messages. Subset Sum properties address target reachability, agreement between target states and positive verdicts, output exclusivity, and cleanup of intermediate objects. Binary Adder properties address sum-bit completeness, final-carry availability, input cleanup, carry cleanup, and arithmetic agreement. Source-level and workflow-level evidence remain distinct throughout the evaluation. Source-level properties are checked on the available KPQ/SPIN/NuSMV translations of the initial and refined sources. Workflow-level properties are checked on finite NuSMV abstractions generated from selected simulation traces. Structural invariants describe stable relations among branches, objects, or arithmetic components, while LTL/CTL formulas express reachability, response, cleanup, and exclusivity conditions. The correspondence between source-level and workflow-level evidence is summarized in Table 8. Table 9 records the input, output, and purpose of each phase in the evaluation protocol. The two experimental levels are connected through the materialized executable sources. Source-level analysis compares the initial and refined or indexed specifications, whereas workflow-level analysis evaluates the representatives produced by candidate generation and selection. This organization relates modelling choices, execution evidence, and temporal verification within a single experimental workflow.

**Table 9.** Evaluation protocol retained across the three benchmarks.

| Phase | Input | Output | Used for |
|---|---|---|---|
| Source simulation | initial/refined source | steps, final objects, residuals | behavioural comparison |
| Source checking | KPQ/LTL properties | SPIN/NuSMV outcomes | safety and liveness evidence |
| Candidate execution | generated `.kpl` sources | logs and real metrics | representative selection |
| Trace checking | selected traces | LTL/CTL truth values | workflow-level verification |

**Table 10.** Selected source-level simulation results for the initial and refined Broadcasting sources. The two column groups list the available cases independently.

| Initial source | | | | Refined source | | | |
|---|---|---|---|---|---|---|---|
| Case | $k$ | $\theta$ | Avg. steps | Case | $k$ | Avg. steps | Final status |
| 1 | 3 | 10 | 4.70 | 1 | 3 | 9.10 | complete, 3/3 stopped |
| 2 | 5 | 10 | 6.10 | 2 | 5 | 9.60 | complete, 5/5 stopped |
| 3 | 10 | 10 | 11.40 | 3 | 10 | 13.30 | complete, 10/10 stopped |
| 4 | 15 | 10 | 16.40 | 4 | 15 | 17.80 | complete, 15/15 stopped |
| 5 | 20 | 10 | 20.90 | 5 | 20 | 20.00 | partial, 17–18/20 stopped |
| 6 | 30 | 10 | 31.10 | 6 | 30 | 20.00 | partial, 17–18/30 stopped |
| 8 | 50 | 10 | 51.00 | 8 | 50 | 20.00 | partial, 17–18/50 stopped |
| 10 | 100 | 10 | 100.60 | 9 | 75 | 20.00 | partial, 18/75 stopped |

## 5 Source-Level Simulation and Property Results

This section reports the direct source-level campaign and compares the standard and indexed or refined sources before the external candidate-generation workflow is applied. Ten kPWorkbench runs were performed for each source-case pair in the source-level simulation campaign. The analysis combines execution metrics, source-level property checks, and representative trace-level property groups in order to relate observable behaviour to the corresponding verification evidence. Broadcasting presents a direct contrast between the initial communication structure and the refined completion mechanism. In the initial source, message delivery is combined with receiver-side matching conditions, while the refined source uses direct return rules and explicit stopped markers. Table 10 reports selected execution measurements for the initial and refined Broadcasting sources. The two column groups summarize the available cases independently and should not be interpreted as row-wise paired configurations. The results show that the refined representation records completion through stopped-family markers and supports direct inspection of cleanup behaviour. For larger instances, the fixed execution window produces partial stopped-family coverage, which is reported explicitly in the final status column. Subset Sum compares exhaustive branch generation with the pruning-aware target-sum source. The two variants represent related search tasks: the initial source exposes the complete partition-style branch space, whereas the pruned source evaluates target-sum reachability while limiting branch extension under positive weights. Table 11 reports the branch counts and result patterns for the exhaustive and pruning-

**Table 11.** Source-level comparison for Subset Sum.

| Case | $n$ | $K$ | Initial | Pruned | Reduction | Pruned $t/f$ | Verdicts |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 3 | 8 | 5 | 37.50% | 2/3 | yes/yes |
| 2 | 4 | 5 | 16 | 9 | 43.75% | 2/7 | yes/yes |
| 3 | 5 | 7 | 32 | 16 | 50.00% | 3/13 | no/yes |
| 4 | 6 | 10 | 64 | 32 | 50.00% | 5/27 | no/yes |
| 5 | 8 | 18 | 256 | 135 | 47.27% | 14/121 | yes/yes |
| 6 | 10 | 27 | 1024 | 512 | 50.00% | 40/472 | no/yes |

**Table 12.** Initial symbolic addition vs. indexed Binary Adder.

| Case | input pos. | repr. value | BA steps | Initial final form | BA final form |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | canonical | canonical |
| 2 | 3 | 3 | 4 | canonical | canonical |
| 3 | 3 | 8 | 4 | non-canonical | canonical |
| 4 | 3 | 16 | 4 | non-canonical | canonical |
| 5 | 7 | 255 | 8 | canonical | canonical |
| 6 | 7 | 256 | 8 | non-canonical | canonical |
| 7 | 15 | 256 | 16 | non-canonical | canonical |
| 8 | 31 | $2^{32}$ | 32 | non-canonical | canonical |

aware Subset Sum sources. Here, $t/f$ denotes the numbers of retained terminal branches associated with true and false target outcomes, respectively, while the two verdict entries refer to the initial and pruning-aware sources. Across the reported cases, the pruning-aware source retains approximately 37%–50% fewer terminal branches than the exhaustive source. The result pattern is interpreted with respect to the semantics of each variant, since the initial source provides a partition-style baseline and the pruned source evaluates target-sum reachability. Binary Adder compares symbolic addition through object multiplicities with an indexed full-adder source. The indexed representation exposes carry propagation and produces explicit binary output objects, whereas the symbolic source preserves value through rewriting rules of the form $2a_i \to a_{i+1}$. The symbolic and indexed Binary Adder executions are compared in Table 12. The indexed source may require additional execution steps because carry propagation is represented explicitly at each bit position. This structure also provides canonical binary output and supports local properties concerning sum bits, carries, and cleanup. The simulation campaign was complemented by source-level KPQ/LTL checking. The properties were translated through the available kPWorkbench verification workflow and evaluated with SPIN and, where the corresponding runs completed, NuSMV. Table 13 reports the source-level Broadcasting checks for the initial and refined sources. In this and the following source-level tables, N/A indicates that the corresponding source variant does not expose a direct object-level predicate for the property, whereas N.C. indicates that the NuSMV execution did not complete within the available run window. After correction of the verifier translations, both SPIN and NuSMV confirm threshold reachability for the initial and refined Broadcasting sources. The property outcomes distinguish the two completion interpretations: the initial source reaches sta-

**Table 13.** Source-level property checking for Broadcasting.

| Property | Source-level check | Init. SPIN | Init. NuSMV | Ref. SPIN | Ref. NuSMV |
|---|---|---|---|---|---|
| Threshold reachability | `F(c1.c >= 10)` | TRUE | TRUE | TRUE | TRUE |
| Response of $a_1$ by $b_1$ | `G(c21.a1>0 -> F c1.b1>0)` | TRUE | TRUE | N/A | N/A |
| Cleanup after threshold | `G(c1.c>=10 -> c2i.ai=0)` | FALSE | FALSE | TRUE | TRUE |
| Stopped-family evidence | `F(all stopped_i)` | N/A | N/A | TRUE | TRUE |
| No residual messages | `G(completion -> no active a_i)` | FALSE | FALSE | TRUE | TRUE |

**Table 14.** Source-level property checking for Subset Sum.

| Property | Source-level check | Initial SPIN | Pruned SPIN |
|---|---|---|---|
| Target/success reachability | `F(t1.t > 0)` or equivalent marker | TRUE | TRUE |
| Positive-verdict agreement | `G(t0.yes>0 -> t1.t>0)` | TRUE | TRUE |
| Terminal marker reachability | `F(t2.x > 0)` | FALSE | FALSE |
| Output exclusivity | `G !(yes_out & no_out)` | TRUE | TRUE |
| Residuals, halting, and steps | trace-level metadata | N/A | N/A |

ble blocked configurations with residual active objects in some runs, whereas the refined source records completion through stopped markers and satisfies the corresponding cleanup properties. The source-level SPIN outcomes for the initial and pruning-aware Subset Sum sources are reported in Table 14. For Subset Sum, SPIN confirms the directly expressible source-level relations, including target reachability, positive-verdict agreement, and output exclusivity. The pruning-aware source retains these checked relations while reducing the explored branch structure. NuSMV checking of the partition source remained outside the completed execution window and is therefore reported separately from the available SPIN evidence. Table 15 reports the available checks for the symbolic and indexed Binary Adder sources. For Binary Adder, the indexed source exposes object-level predicates for sum-bit completion, carry availability, carry cleanup, and arithmetic agreement. The initial symbolic source remains value-oriented, and the SPIN and NuSMV outcomes for sum-bit completeness reflect the different asynchronous translations used by the two verification back ends. After correction of the verifier mappings, the indexed-source results agree for the directly expressible properties and support property-oriented analysis of the full-adder structure. The workflow-level NuSMV abstractions use Boolean pred-

**Table 15.** Source-level property checking for Binary Adder.

| Property | Source-level check | Init. SPIN | Init. NuSMV | Indexed SPIN | Indexed NuSMV |
|---|---|---|---|---|---|
| Natural halting | explicit halt predicate | N/A | N/A | N/A | N/A |
| Sum-bit completeness | `F(output sum objects)` | TRUE | TRUE | TRUE | TRUE |
| Final-carry availability | `F(carry4_0 > 0)` | N/A | N/A | TRUE | N.C. |
| Input cleanup | `F(no input bit objects)` | TRUE | TRUE | TRUE | TRUE |
| Carry cleanup | `F(no intermediate carry)` | N/A | N/A | TRUE | TRUE |
| Arithmetic agreement | expected output-bit encoding | N/A | N/A | TRUE | TRUE |

**Table 16.** Representative workflow-level temporal property groups.

| Problem | Property group | Representative formula |
|---|---|---|
| Broadcasting | natural halting | `LTLSPEC F halted` |
| Broadcasting | cleanup after threshold | `G(threshold_reached -> F no_active_messages)` |
| Broadcasting | no residual messages | `G(halted -> no_residual_messages)` |
| Subset Sum | target reachability | `LTLSPEC F target_reached` |
| Subset Sum | positive-verdict agreement | `G(positive_verdict -> target_reached)` |
| Subset Sum | output exclusivity | `G !(yes_out & no_out)` |
| Binary Adder | sum-bit completeness | `G(halted -> all_sum_bits_ready)` |
| Binary Adder | carry cleanup | `G(halted -> no_intermediate_carry)` |
| Binary Adder | arithmetic agreement | `G(halted -> arithmetic_ok)` |

icates derived from concrete simulation traces. Table 16 summarizes the main property groups evaluated after representative selection, while the complete generated artefacts contain the expanded declarations and formula instances. The source-level tables describe the behaviour observed before the external workflow is applied. Broadcasting distinguishes stable blocking from completion recorded through explicit markers, Subset Sum quantifies the effect of target-aware pruning, and Binary Adder distinguishes value preservation from canonical bit-level output. The property tables complement these observations by identifying the predicates that are directly available for each source variant and verification back end. The workflow-level representatives are selected according to the structure of each benchmark. Broadcasting uses static candidate ranking because the refined source and the case parameters already define the relevant instances. For Subset Sum, descriptor choices concern item ordering and pruning trajectories, which influence retained branches and execution stages. For Binary Adder, the candidate descriptors concern grouping and carry organization, which influence the regularity of the generated source and its property structure. The reported fitness values are computed from real kPWorkbench executions after the generated sources have been materialized and their logs parsed. Proxy scores are used to organize candidate search, while real execution metrics determine the reported representatives. NuSMV verification is subsequently applied to the selected concrete traces. Table 17 aggregates the primary verification campaign over the selected represen-

**Table 17.** Workflow-level summary across selected representatives.

| Problem | Cases | Real runs | NuSMV properties | False | Status |
|---|---|---|---|---|---|
| Broadcasting | 6 | 375 | 78 | 0 | PASS |
| Subset Sum | 6 | 468 | 66 | 0 | PASS |
| Binary Adder | 6 | 360 | 138 | 0 | PASS |
| Total | 18 | 1203 | 282 | 0 | PASS |

**Table 18.** Subset Sum control audit using non-vacuous success properties.

| Property | Selected | Unselected controls |
|---|---|---|
| F *reached_target* | TRUE | FALSE |
| AF *reached_target* | TRUE | FALSE |
| F *positive_global_verdict* | TRUE | FALSE |
| AF *positive_global_verdict* | TRUE | FALSE |

tatives. A separate Subset Sum control audit, reported below, is not included in the 282-property total. For Subset Sum, the six selected representatives were additionally checked against four non-vacuous success properties: F *reached_target*, AF *reached_target*, F *positive_global_verdict*, and AF *positive_global_verdict*. All four properties hold for every selected representative. The same checks were applied to three lower-fitness unselected candidates: `case02_cand001`, `case03_cand001`, and `case04_cand004`; all four properties are false for each control candidate. Together with the 11 standard properties, this gives 15/15 true outcomes for each selected Subset Sum representative. Each unselected control satisfies the 11 standard structural and consistency properties but fails the four non-vacuous success properties, yielding 11/15 true outcomes. Overall, the section connects direct source-level simulation, translated-source checking, and temporal verification of concrete traces. The additional Subset Sum controls show that the non-vacuous reachability and positive-verdict properties distinguish the selected representatives from sampled lower-fitness candidates, while both groups retain the standard structural properties.

## 6 Discussion, Conclusions, and Future Work

The results should be interpreted primarily as a comparison of modelling and verification choices rather than as a uniform comparison of execution speed. The refined Broadcasting source separates threshold reachability from cleanup and records completion through explicit stopped markers. The pruning-aware Subset Sum source reduces the explored branch structure while retaining the directly checked safety and verdict relations. The indexed Binary Adder may require additional steps because carry propagation is represented explicitly, but this representation exposes the sum and carry structures needed for canonical output and local arithmetic properties. Refinement and indexation therefore contribute principally to semantic explicitness, parameterization, and verification accessibility; they do not remove the intrinsic computational complexity of the underlying benchmark. The source-level experiments also illustrate the importance of verifier-specific translations. During model development, differences

between SPIN and NuSMV were associated with object mappings, execution semantics, and finite bounds. For example, the Broadcasting counter bound must allow a valid threshold overshoot: when $\theta = 10$ and $k = 3$, a configuration containing $9c$ may receive three additional counter objects and therefore requires the finite abstraction to represent $12c$. In Binary Adder, output objects must likewise be mapped to the intended compartments and bit positions. These observations motivate keeping the executable source, its concrete simulation evidence, and the verifier-specific artefact as identifiable stages of the experimental record. The external layer has a complementary role. It explores or organizes descriptor-level alternatives before executable evaluation but does not itself provide formal correctness evidence. Broadcasting uses deterministic candidate generation, whereas Subset Sum and Binary Adder use the genetic configurations reported in Table 6. In all three cases, descriptor-level proxy ranking is followed by source materialization, kPWorkbench execution, metric extraction, and case-level representative selection. NuSMV is applied only after these stages. This separation prevents a high proxy score from being interpreted as either a successful simulation or a satisfied temporal property. The Subset Sum control audit gives a concrete example of the discriminating role of the temporal properties. Each selected representative satisfies all 15 examined properties, whereas the sampled lower-fitness candidates `case02_cand001`, `case03_cand001`, and `case04_cand004` satisfy the 11 structural and consistency properties but fail the four non-vacuous target-reachability and positive-verdict properties. The distinction is therefore not produced by changing the property set between selected and unselected candidates, but by applying the same success-oriented temporal requirements to behaviours with different observed outcomes. The verification claims remain deliberately trace-level. The workflow-level NuSMV models are finite abstractions constructed from concrete kPWorkbench simulation evidence and from the predicates required by the reported formulas. They validate the selected observed behaviours but do not constitute a complete semantic translation of the full kP-Lingua language. Source-level SPIN/NuSMV checks and workflow-level trace abstractions must consequently be read as complementary evidence rather than as interchangeable verification layers. In summary, the study provides an integrated verification-aware methodology connecting indexed kernel P-system modelling, benchmark-specific external candidate generation, real executable evaluation, execution-derived selection, and temporal checking across communication, combinatorial, and arithmetic benchmarks. Its principal contribution is the explicit separation of descriptor-level search, executable behavioural evidence, and formal property outcomes while retaining a reproducible link between them. Future work includes improving the Promela sources used for verification, examining query-based verification in `kPWorkbench` or kP-Query, developing a closer correspondence between indexed `kP-Lingua` specifications and verifier-specific temporal models, extending the control experiments to larger candidate samples, and exploring complementary probabilistic model-checking tools such as MC2 [27] for stochastic refined variants.

## A Human-Readable kP Encodings of the Benchmarks

This appendix summarizes the computational interpretation of the three kP encodings independently of their executable kP-Lingua syntax. The purpose is to make explicit how the objects, compartments, and principal rule families represent each benchmark. The complete executable specifications remain available with the experimental artefacts.

*Broadcasting.* The Broadcasting system contains one central compartment and a family of receiver compartments. For each family $i$, the central compartment initially contains an object $b_i$. Below threshold $\theta$, $b_i$ can be activated as $a_i$ and transmitted to a receiver. Receiver $i$ is conceptually associated with marker $x_i$. In the initial source, receivers share a common type and target selection is nondeterministic; an active object $a_i$ can therefore reach a receiver marked $x_j$, $j \neq i$, where the guarded acknowledgement rule for family $i$ is unavailable. A successful acknowledgement returns $b_i$ to the central compartment and contributes $i$ copies of counter object $c$. Once the multiplicity of $c$ reaches the threshold, the execution enters its cleanup regime. The refined encoding replaces the problematic receiver-side matching behaviour by direct return rules and records completed families explicitly through `stopped_i` markers. Consequently, threshold reachability, cleanup, and completed-family evidence can be expressed as separate properties.

*Partition/Subset Sum.* The initial Partition/Subset Sum encoding represents the include/exclude decisions for $n$ items through parallel branch generation. At stage $i$, a branch creates alternatives corresponding to inclusion or exclusion of item $v_i$; an object $b_i$ records membership of $v_i$ in the selected subset. After all $n$ stages, the exhaustive construction can expose up to $2^n$ candidate branches. For Subset Sum, the instance additionally contains a prescribed target $K$. The pruning-aware encoding associates a partial selected sum with each active branch. Because the benchmark uses positive weights, an inclusion transition that would increase the partial sum beyond $K$ cannot lead to a solution and is therefore not extended. A terminal branch whose accumulated value reaches $K$ generates the positive target/verdict evidence; other retained terminal branches contribute negative evidence. The output rules enforce the intended yes/no interpretation and support exclusivity and verdict-consistency checks. Thus pruning changes the explored branch structure while preserving the target-sum semantics checked in the reported experiments.

*Binary Adder.* The symbolic Binary Adder baseline represents numerical value by object multiplicities. A rule $2a_i \rightarrow a_{i+1}$ implements a binary carry by replacing two objects of value $2^i$ with one object of value $2^{i+1}$. This representation preserves numerical value but does not by itself expose a canonical sum bit at every position. The indexed full-adder variant instead represents both input bits explicitly. At position $i$, objects $x_{i,0}$ or $x_{i,1}$ encode the first input bit, $y_{i,0}$ or $y_{i,1}$ encode the second, and $carry_{i,0}$ or $carry_{i,1}$ encode the incoming carry. The eight combinations of these three Boolean inputs are represented by indexed full-adder rules that generate the corresponding $sum_{i,0/1}$ object and $carry_{i+1,0/1}$. Processing proceeds across the indexed bit positions until all sum

bits and the final carry have been determined. This explicit representation makes sum completeness, final-carry availability, input cleanup, working-carry cleanup, and arithmetic agreement directly available as verification predicates.

## References


1. Păun, Gh.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1690
2. Păun, Gh.: Membrane Computing: An Introduction. Springer, Berlin (2002). https://doi.org/10.1007/978-3-642-56196-2
3. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010). https://global.oup.com/academic/product/the-oxford-handbook-of-membrane-computing-9780199556670
4. Rozenberg, G., Bäck, T., Kok, J.N. (eds.): Handbook of Natural Computing. Springer, Berlin (2012). https://doi.org/10.1007/978-3-540-92910-9
5. Gheorghe, M., Ceterchi, R., Ipate, F., Konur, S., Lefticaru, R.: Kernel P systems: From modelling to verification and testing. Theoretical Computer Science 724, 45–60 (2018). https://doi.org/10.1016/j.tcs.2017.12.010
6. Kernel-P-Systems: kPWorkbench repository. https://github.com/Kernel-P-Systems/kPWorkbench
7. Iacob, S., Ipate, F., Niculescu, I.M., Konur, S.: Some new developments in kP-Lingua. Manuscript and kPWorkbench resources (2025). https://github.com/Kernel-P-Systems/kPWorkbench
8. Research Group on Natural Computing: P-Lingua and membrane computing software resources. https://www.gcn.us.es/
9. Ciobanu, Gh., Păun, Gh., Pérez-Jiménez, M.J. (eds.): Applications of Membrane Computing. Springer, Berlin (2006). https://doi.org/10.1007/3-540-29937-8
10. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A., Romero-Campero, F.J.: A linear solution for QSAT using membrane creation. Theoretical Computer Science 371(1–2), 1–11 (2007). https://doi.org/10.1016/j.tcs.2006.11.007
11. Alhazov, A., Freund, R., Ivanov, S., Verlan, S.: P systems with limited resources. Theoretical Computer Science 529, 1–17 (2014). https://doi.org/10.1016/j.tcs.2014.01.005
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA (1999). https://mitpress.mit.edu/9780262032704/model-checking/
13. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge, MA (2008). https://mitpress.mit.edu/9780262026499/principles-of-model-checking/
14. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
15. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986). https://doi.org/10.1145/5397.5399
16. Kozen, D.: Results on the propositional $\mu$-calculus. Theoretical Computer Science 27, 333–354 (1983). https://doi.org/10.1016/0304-3975(82)90125-6

17. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997). https://doi.org/10.1109/32.588521
18. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2004). https://spinroot.com/spin/Doc/Book_extras/
19. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model checker. International Journal on Software Tools for Technology Transfer 2, 410–425 (2000). https://doi.org/10.1007/s100090050046
20. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47
21. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H. et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer (2010). https://doi.org/10.1007/978-3-642-16612-9_11
22. Bobe, R.T., Konur, S.: Probabilistic verification of robot controllers using numerical P systems models. Manuscript (2025). https://github.com/Kernel-P-Systems/kPWorkbench
23. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972). https://doi.org/10.1007/978-1-4684-2001-2_9
24. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman (1979). https://dl.acm.org/doi/book/10.5555/578533
25. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press (1975). https://mitpress.mit.edu/9780262581110/adaptation-in-natural-and-artificial-systems/
26. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley (1989). https://dl.acm.org/doi/book/10.5555/534133
27. Gilbert, D., Donaldson, R.: A Monte Carlo model checker for probabilistic ltl with numerical constraints (2008)
28. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press (1998). https://mitpress.mit.edu/9780262631853/an-introduction-to-genetic-algorithms/
29. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing, 2nd edn. Springer (2015). https://doi.org/10.1007/978-3-662-44874-8
30. Whitley, D.: A genetic algorithm tutorial. Statistics and Computing 4, 65–85 (1994). https://doi.org/10.1007/BF00175354
31. Cabarle, F.G.C.: Thinking about spiking neural P systems: some theories, tools, and research topics. Journal of Membrane Computing 6, 148–167 (2024). https://doi.org/10.1007/s41965-024-00147-y
32. Andreu-Guzmán, J.A., Orellana-Martín, D., Valencia-Cabrera, L.: Dissecting OLMS membrane algorithms: understanding the role of communication and evolutionary operators in optimization strategies. Journal of Membrane Computing 7, 475–520 (2025). https://doi.org/10.1007/s41965-025-00200-4
33. Kuczik, A., Vaszil, Gy.: Non-cooperative polymorphic P systems and parallel communicating ET0L systems. Journal of Membrane Computing 7, 312–325 (2025). https://doi.org/10.1007/s41965-025-00185-0

# A polynomial-time solution to the latin square completion problem by means of P systems with active membranes

Jerónimo Crespo[1] and David Orellana-Martín[2,3][0000−0002−2892−6775]

[1] Departamento de Álgebra, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain
jercrealm@alum.us.es

[2] Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain

[3] SCORE Laboratory, I3US, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain
dorellana@us.es

**Abstract.** Latin squares are classical combinatorial structures with applications in experimental design, coding theory, and several areas of discrete mathematics. A particular problem related to this structure, called the Latin Square Completion problem asks whether a partial Latin square can be completed so that every symbol occurs exactly once in each row and each column. This problem is known to be **NP**-complete. In this work, an explicit solution to this problem by means of a uniform family of recognizer P systems with active membranes and elementary membrane division is provided, following the usual generation, checking and output stages workflow. We show that the solution operates in polynomial time and its description remains polynomial in the order of the input square.

**Keywords:** Latin squares · P systems · Active membranes · Membrane division.

## 1 Introduction

Latin squares are classical combinatorial structures and they are widely studied. They are the base, for instance, for the theory of combinatorial design, have been studied from algebraic, combinatorial and computational points of view. Latin squares have many applications in different areas such as experimental design, coding theory, planning and other assignment problems [2, 9].

Given a partial Latin square; that is, a Latin square where not every cell has a number assigned, it has been proved that knowing if there exists a completion of such a square is an **NP**-complete problem [1]. Such a decision problem is called Latin square completion problem, or `LSC` problem. Different versions of the problem were studied in [5]. For solving this problem (either the search or the decision problem), different methods have been taken into account, such

as combinatorial formulation and exact methods, reductions to graph-coloring problems, memetic algorithms, local search and evolutionary computation [5, 6, 8, 10].

Natural computing is a branch of computer science where different models of computation arise from the inspiration of different systems of nature. In particular, in 1998, Gheorghe Păun introduces *membrane systems* or *P systems* [11, 13, 14], that are devices inspired by the structure and behavior of living cells. In particular, these systems take into account the internal topology of cells and have different regions delimited by membranes and there exist rules that make the objects and membranes evolve in parallel time through their execution.

In the framework of membrane computing, a particular type of P systems, called P systems with active membranes, introduced in [12], are very relevant for solving computationally hard problems because of a very interesting ingredient: division rules, that allow the system to generate an exponential amount of regions in polynomial time, taking advantage of the inherent parallel behavior of these devices. This capability has been used multiple times to solve **NP**-complete problems [16]. In [18], a computational complexity theory in membrane computing is described by means of recognizer P systems as devices capable of solving decision problems.

In this sense, different **NP**-complete problems, such as `SAT`, `SubsetSum` and `PARTITION` problems have been efficiently solved, among others [7, 17, 24]. These solutions usually adopt the following pattern: First, all the possible assignments are generated; Then, the tentative solutions are checked; Finally, if at least one solution has been proved to be correct, then an object `yes` is sent to the environment, and an object `no` otherwise [21]. Related grid-based constraint problems, such as Sudoku, have also been solved by means of membrane systems [4, 3, 23].

While the results of the papers that introduce a solution to an **NP**-complete problem are to show that all the problems in **NP** can be solved by means of families of recognizer P systems of such type, the idea of this work is not that one, but to explore and provide an explicit solution to the structure of Latin squares by means of a uniform family of recognizer P systems with active membranes working in $O(n^3)$ computational steps. Besides, Latin Square Completion differs from `SAT` and other decision problems since it deals with two overlapping families of global constraints (row and column), and the fixed values of the partial Latin square must be preserved. Besides, the exponential generation of the possible solutions is not bounded by $2^n$ but, in the worst case, by $n^{n^2}$.

The rest of the work is organized as follows: In Section 2, some useful concepts for the rest of the paper are defined. Section 3 is devoted to provide the solution to the Latin square completion problem, providing an overview of the computation. In the next section, some hints about the correctness of the construction are given, besides the computational resources needed by the P systems from the solution. In Section 5, an explicit example is provided in a graphical way for the sake of simplicity. Finally, some conclusions and open research lines are detailed in the last section.

## 2 Preliminaries

In this section, some necessary concepts for the rest of the paper can be found. For a comprehensive overview of formal languages and membrane systems, we refer the reader to [22] and [15], respectively.

### 2.1 Latin squares

A Latin square of order $n$ is an $n \times n$ array whose entries belong to $\{1, \ldots, n\}$ and in which each symbol appears exactly once in each row and exactly once in each column [9]. Consequently, each row and each column is a permutation of $\{1, \ldots, n\}$.

A partial Latin square of order $n$ is defined $L = (n, N)$, $N \in \{1, \ldots, n\} \times \{1, \ldots, n\} \times \{1, \ldots, n\}$, where $(i, j, k) \in N$ means that the cell $(i, j)$ contains the symbol $k$, subject to the following restrictions:

$$(i, j, k), (i, j, k') \in N \Rightarrow k = k',$$

$$(i, j, k), (i, j', k) \in N \Rightarrow j = j', \text{ and}$$

$$(i, j, k), (i', j, k) \in N \Rightarrow i = i'.$$

$C : \{1, \ldots, n\} \times \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a completion of a partial Latin square $L$ if $C$ is a Latin square and $C(i, j) = k$ if $(i, j, k) \in N$.

The Latin square completion problem is a decision problem that, given a partial Latin square $L$ of order $n$, the question is if there exists a Latin square of order $n$ that completes $L$. This problem can be seen in the literature as the `LSC` problem.

A complete Latin square provides a certificate that can be verified in polynomial time. Moreover, Colbourn proved that the `LSC` problem is **NP**-complete [1].

### 2.2 P systems with active membranes

Cell-like P systems with active membranes having associated electrical charges with membranes were first introduced by Gh. Păun [12]. One of the main attractions of these membrane systems is the ability to create an exponential workspace, expressed in terms of number of objects and number of membranes, in linear time.

**Definition 1.** *A P* system with active membranes *of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where:*

1. *$\Gamma$ is a finite alphabet.*
2. *$\mu$ is a rooted tree.*
3. *$\mathcal{M}_1, \ldots, \mathcal{M}_q$ are multisets over $\Gamma$.*
4. *$\mathcal{R}$ is a finite set of rules ($H$ denotes the set $\{1, \ldots, q\}$) of the following forms:*
   *(a) $[\, a \to u\,]_h^\alpha$, for $h \in H$, $\alpha \in \{+, -, 0\}$, $a \in \Gamma$, $u \in M(\Gamma)$* (object-evolution rules).

*(b)* $a[\ ]_h^{\alpha_1} \rightarrow [b]_h^{\alpha_2}$, *for* $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$ (send–in communication rules).
*(c)* $[a]_h^{\alpha_1} \rightarrow [\ ]_h^{\alpha_2} b$, *for* $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$ (send–out communication rules).
*(d)* $[a]_h^{\alpha} \rightarrow b$, *for* $h \in H$, $\alpha \in \{+, -, 0\}$, $a, b \in \Gamma$ (dissolution rules).
*(e)* $[a]_h^{\alpha_1} \rightarrow [b]_h^{\alpha_2} [c]_h^{\alpha_3}$, *for* $h \in H$, $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$, $a, b, c \in \Gamma$ (division rules for elementary membranes).

5. $i_{out} \in H \cup \{0\}$.

A *P system with active membranes* $\Pi = (\Gamma, \mu, , \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, of degree $q \geq 1$ can be viewed as a set of $q$ membranes injectively labelled by $1, \ldots, q$ with electrical charges (positive $+$, negative $-$ or neutral 0) associated to them, starting with neutral polarization, arranged in a hierarchical structure $\mu$ given by a rooted tree whose root is called the *skin membrane* of the system and with an environment labelled by 0 such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are multisets over the *working alphabet* $\Gamma$ representing the objects initially placed in the $q$ membranes of the system; (b) $\mathcal{R}$ is the set of rules that allow to evolve the system; and (c) $i_{out} \in \{0, 1, 2, \ldots, q\}$ represents a distinguished *region* which will contain the output of the system (the term *region* is used as above).

Notice that P systems with active membranes have some important features: (a) they use three electrical charges; (b) the polarization of a membrane, but not the label, can be modified by the application of a rule; (c) the rules are non-cooperative (the corresponding left-hand side consists of only one symbol), and (d) there are no priorities among rules.

An *instantaneous description* or a *configuration* at an instant $t$ of a P system with active membranes $\Pi = (\Gamma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ is described by the membrane structure at instant $t$ with the corresponding polarizations, and all multisets of objects over $\Gamma$ associated with all the membranes present in the system. The *initial configuration* of the system is $(\mu, \mathcal{M}_1, \cdots, \mathcal{M}_q)$.

An object-evolution rule $[a \rightarrow u]_h^{\alpha}$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant $t$ if the following holds: (a) a membrane labelled by $h$ is in $\mathcal{C}_t$ and its electrical charge is $\alpha$; and (b) object $a$ is contained in such membrane. When applying such a rule, object $a$ is consumed and the objects specified by multiset $u$ will be placed in that membrane $h$.

A send-in rule $a[\ ]_h^{\alpha_1} \rightarrow [b]_h^{\alpha_2}$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant $t$ if the following holds: (a) a membrane labelled by $h$, different from the skin membrane, is in $\mathcal{C}_t$ and its electrical charge is $\alpha_1$; and (b) object $a$ is contained in the region immediately outside the membrane $h$, the parent $p(h)$ of that membrane. When applying such a rule, object $a$ is consumed, object $b$ will be placed in the same membrane $h$ and the polarization of such membrane $h$ will change to $\alpha_2$.

A send-out rule $[\ a\ ]_h^{\alpha_1} \rightarrow b[\ ]_h^{\alpha_2}$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant $t$ if the following holds: (a) a membrane labelled by $h$, is in $\mathcal{C}_t$ and its electrical charge is $\alpha_1$; and (b) object $a$ is contained in such membrane $h$. When applying such a rule, object $a$ is consumed, object $b$ will be placed in the region

immediately outside the membrane $h$, the parent $p(h)$ of that membrane, and the polarization of such membrane $h$ will change to $\alpha_2$.

A dissolution rule $[\,a\,]_h^\alpha \to b$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant $t$ if the following holds: (a) a membrane labelled by $h$ is in $\mathcal{C}_t$ and its electrical charge is $\alpha$; and (b) object $a$ is contained in such membrane $h$. When applying such a rule, object $a$ is consumed and the membrane is dissolved. After dissolving such membrane $h$, all objects and membranes previously present in it become elements of the contents of the immediately upper membrane which has not been dissolved, except object $a$ triggering the rule that evolves to $b$.

A division rule for elementary membrane $[\,a\,]_h^{\alpha_1} \to [\,b\,]_h^{\alpha_2}\ [\,c\,]_h^{\alpha_3}$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant $t$ if the following holds: (a) an elementary membrane labelled by $h$, different from the skin membrane and the output membrane, is in $\mathcal{C}_t$ and its electrical charge is $\alpha_1$; (b) $h \neq i_{out}$; and (c) object $a$ is contained in such membrane $h$. When applying such a rule, object $a$ is consumed and the membrane is divided into two membranes with the same label $h$, maybe of different polarizations $\alpha_2$ and $\alpha_3$; the object $a$ specified in the rule is replaced in the two new membranes by possibly new objects $b$ and $c$, respectively, and the rest of the objects is replicated in both new membranes.

If the polarization of any of the membranes involved in the rule is not indicated, then that polarization is neutral.

### 2.3 Recognizer P systems

In membrane computing, decision problems are solved through families of recognizer P systems. In this work, we follow the standard definition of uniform families and polynomial-time solutions to decision problems introduced in [18, 19]. In this sense, each P system $\Pi(n)$ from a family $\{\Pi(n) : n \in \mathbb{N}\}$ is associated with all the inputs of size $n$, and the corresponding input is encoded as an input multiset. Both the construction of such a P system and the encoding of the input instance are required to be computable in polynomial time.

A recognizer P system $(\Pi, \Sigma, i_{in})$ is an extension of any type of P system where $\Pi$ is a P system, $\Sigma$ is the input alphabet, that is a subset of the working alphabet $\Gamma$, and $i_{in}$ is the input region where a multiset of objects $m$ from $\Sigma$ will be included in the initial configuration apart from the usual $\mathcal{M}_{i_{in}}$, denoted by $\Pi + m$. Let $X = (I_X, \theta_X)$ be a decision problem, and let $\mathcal{R}$ be a family of P systems. Then, a polynomial-time uniform solution to $X$ is a family of recognizer P systems $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$ from $\mathcal{R}$ if the following happens:

1. There exists a pair of polynomial-time computable functions $(cod, s)$ such that for each $u \in I_X$, $cod(u) \in M(\Sigma)$, where $M(\Sigma)$ is the set of all multisets whose elements belong to $\Sigma$, $s(u) \in \mathbb{N}$ and $s^{-1}(n) \subseteq I_X$.
2. For each instance $u \in I_X$, all the computations of $\Pi(s(u)) + cod(u)$ halt and all of them return either the same object `yes` or an object `no` (but not both) at the final step of the computation.
3. All the construction of the P system and the working time of the system must be polynomial-time computable.

In the case of recognizer P systems with active membranes, the original tuple is transformed into

$$\Pi(n) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out}).$$

A more detailed specification can be found in [18, 19].

## 3 Solving the Latin Square Completion Problem with Active Membranes

The Latin Square Completion problem belongs to **NP**. Moreover, it is known that $\mathbf{NP} \subseteq \mathbf{PMC}_{\mathcal{AM}}$ where $\mathbf{PMC}_{\mathcal{AM}}$ denotes the class of decision problems solvable in polynomial time by uniform families of recognizer P systems with active membranes [12, 20]. Therefore, the existence of a polynomial-time solution to the Latin Square Completion problem in this framework follows directly from the previous result. However, an explicit solution to this problem has not been provided before in this framework.

Given a partial Latin square $L = (n, N)$ of size $n$, $s(L) = n$ and $cod(L) = \{K_{ij}^{(0)} : (i, j, K) \in N\} \cup \{X_{ij}^{(0)} : \nexists k : (i, j, k) \in N\}$, where $K_{ij}^{(0)}$ represents a cell with the fixed value $K$, and $X_{ij}^{(0)}$ an empty cell.

We design the following uniform family of recognizer P systems with active membranes $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$ that solves the Partial Latin Square Completion problem (LSC) in polynomial time. More precisely, $\Pi(n)$ solves all the partial Latin squares of size $n$; that is, every $L = (n, N)$. The P system is

$$\Pi(n) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, \mathcal{R}, i_{in}, i_{out}),$$

where:

- The input alphabet is of the form $\Sigma = \{X_{ij}^{(0)} : 1 \leq i, j \leq n\} \cup \{K_{ij}^{(0)} : 1 \leq i, j, K \leq n\}$;
- The working alphabet is $\Gamma = \Sigma \cup \{g_{ij}^k : k \in \{0, ..., n+1\}, (i, j) \in \{1, ..., n\}^2\} \cup \{c_l^k : k \in \{1, ..., n+1\}, l \in \{1, ..., 4n+1\}\} \cup \{f_l^k : k \in \{1, ..., n+1\}, l \in \{1, ..., 4n+1\}\} \cup \{K_{ij}^{\prime(m)} : 1 \leq i, j \leq n, K \in \{1, ..., n\}, m \in 1, ..., n^2\} \cup \{\boxed{K}^{*(m)}(l)_{ij} : 1 \leq i, j \leq n, K \in \{1, ..., n\}, l \in \mathbb{Z}_n, m \in \{1, ..., 2n\}\} \cup \{\bar{K}^{*(m)}(l)_{ij} : 1 \leq i, j \leq n, K \in \{1, ..., n\}, l \in \mathbb{Z}_n, m \in \{1, ..., 2n+1\}\} \cup \{N_k^m : k \in \{1, ..., 4n+1\}, t \in \{1, ..., 2n\}\} \cup \{g, B, Y_0, Y_1, Y_2, Y, F, \texttt{yes}, \texttt{no}\}$;
- $\mu = [[]_2]_1$;
- $\mathcal{M}_1 = \{B\}$, $\mathcal{M}_2 = \{g\}$;
- The input membrane is $i_{in} = 2$ and the output region is $i_{out} = env$.

The set of rules $\mathcal{R}$ is the following:

a) $[g]_2^0 \to [g_{11}^0]_2^+$; $[g_{ij}^k]_2 \to [g_{ij}^{k+1}]_2 \; \forall i, j, 0 \leq k \leq n-1$; $[g_{ij}^n]_2^0 \to [g_{ij}^{n+1}]_2^- \; \forall i, j$; $[g_{ij}^{n+1}]_2^- \to [g_{i(j+1)}^0]_2^+ \; \forall i, j \neq n$; $[g_{in}^{n+1}]_2^- \to [g_{(i+1)1}^0]_2^+ \; i \neq n$; $[g_{nn}^{n+1}]_2^- \to c_0^1[]_2^0$

b) $[X_{11}^{(m)}]_2^+ \to [1_{11}^{\prime(m)}]_2^+ [1_{11}^{(m)}]_2^0$; $[K_{11}^{\prime(m)}]_2^+ \to [(K+1)_{11}^{\prime(m)}]_2^+ [(K+1)_{11}^{(m)}]_2^0$ $1 \leq K \leq n-2, m \in \{1,...,n^2\}$; $[(n-1)_{11}^{\prime(m)}]_2^+ \to [n_{11}^{(m)}]_2^0$ ; $[K_{11}^{(m)}]_2^+ \to [K_{11}^{(m)}]_2^0 1 \leq K \leq n$

c) $[X_{ij}^{(m)}]_2^- \to [X_{i(j-1)}^{(m+1)}]_2^- \; \forall i,j \neq 1$; $[X_{i1}^{(m)}]_2^- \to [X_{(i-1)n}^{(m+1)}]_2^- \; i \neq 1$; $[X_{11}^{(m)}]_2^- \to [X_{nn}^{(m+1)}]_2^-$; $[K_{ij}^{(m)}]_2^- \to [K_{i(j-1)}^{(m+1)}]_2^- \; \forall i,j \neq 1$; $[K_{i1}^{(m)}]_2^- \to [K_{(i-1)n}^{(m+1)}]_2^- \; i \neq 1$; $[K_{11}^{(m)}]_2^- \to [K_{nn}^{(m+1)}]_2^-$

d) $[1_{1j}^{*(m)}(l)]_2^0 \to [\bar{1}_{1j}^{*(m)}(l)]_2^0$; $[\bar{1}_{1j}^{*(m)}(l)]_2^0 \to [\bar{n}_{1j}^{*(m)}(l+1)]_2^+$;
$[\bar{n}_{1j}^{*(m)}(l)]_2^+ \to [\boxed{n}_{1j}^{*(m)}(l)]_2^+ \; \forall j$;
$[\boxed{n}_{1j}^{*(m)}(l)]_2^+ \to [n_{1j}^{*(m)}(l)]_2^0$

e) $[\bar{1}_{1j}^{*(m)}(l)]_2^+ \to []_2^- \; \forall j,m,l \in \mathbb{Z}_{n+1}$; $[K_{1j}^{*(m)}(l)]_2^+ \to [(\boxed{K-1})_{1j}^{*(m)}(l+1)]_2^+ \; \forall j$, $K \in \{2,...,n\}, m \in \{0,...,n\}, l \in \{1,...,n-1\}$; $[\boxed{K}_{1j}^{*(m)}(l)] \to [K_{1j}^{*(m)}(l)]_2^+ \; \forall j$, $K \in \{1,...,n-1\}, m \in \{0,...,n\}, l \in \mathbb{Z}_n$

f) $[c_k^m]_1^0 \to [c_{k+1}^m]_1^0 \; \forall k \in \{0,...,4n-1\}, m \in \{1,...,n\}$; $c_{4n}^m []_2^0 \to [c_{4n+1}^m]_2^- \; m \in \{1,...,n\}$; $[c_{4n+1}^m]_2^- \to c_0^{m+1}[]_2^0 \; m \in \{1,...,n-1\}$; $[c_{4n+1}^n]_2^- \to f_o^1[]_2^0$; $[c_{4n}^m]_1^0 \to [N_{4n+1}^m]_1^0 \; m \in \{1,...,n\}$

g) $[\bar{1}_{1j}^{*(m)}]_2^- \to [1_{nj}^{*(m+1)}]_2^- \; \forall j,m \in \{1,...,n\}$; $[K_{ij}^{*(m)}]_2^- \to [K_{(i-1)j}^{*(m+1)}]_2^- \; \forall j, i \neq 1, m \in \{1,...,n\}, K \in \{1,...,n\}$; $[K_{1j}^{*(m)}]_2^- \to [K_{nj}^{*(m+1)}]_2^- \; \forall j, m \in \{1,...,n\}$, $K \in \{1,...,n\}$; $[K_{1j}^{*(m)}(l)]_2^- \to [F]_2^- \; l \neq 0, K \in \{1,...,n\}, \forall j$; $[F]_2^0 \to []_2^-$

h) $[1_{i1}^{**(m)}(l)]_2^0 \to [\bar{1}_{i1}^{**(m)}(l)]_2^0$; $[\bar{1}_{i1}^{**(m)}(l)]_2^0 \to [\bar{n}_{i1}^{**(m)}(l+1)]_2^+$; $[\bar{n}_{i1}^{**(m)}(l)]_2^+ \to [\boxed{n}_{i1}^{**(m)}(l)]_2^+ \; \forall i$;
$[\boxed{n}_{i1}^{**(m)}(l)]_2^+ \to [n_{i1}^{**(m)}(l)]_2^0$

i) $[\bar{1}_{i1}^{**(m)}(l)]_2^+ \to []_2^- \; \forall j,m$; $[K_{i1}^{**(m)}(l)]_2^+ \to [(\boxed{K-1})_{i1}^{**(m)}(l-1)]_2^+ \; \forall i, K \in \{2,...,n\}, m \in \{0,...,n\}$; $[\boxed{K}_{i1}^{**(m)}(l)] \to [K_{i1}^{**(m)}(l)]_2^+ \; \forall i, K \in \{1,...,n-1\}, m \in \{0,...,n\}, l \in \mathbb{Z}_n$

j) $[f_k^m]_1^0 \to [f_{k+1}^m]_1^0 \; \forall k \in \{0,...,4n-1\}, m \in \{1,...,n\}$; $f_{4n}^m []_2^0 \to [f_{4n+1}^m]_2^- \; m \in \{1,...,n\}$; $[f_{4n+1}^m]_2^- \to f_0^{m+1}[]_2^0 \; m \in \{1,...,n-1\}$; $[f_{4n+1}^n]_2^- \to Y_0[]_2^0$; $[Y_0]_1^0 \to [Y_1]_1^0$; $Y_1[]_2^0 \to [Y_2]_2^0$; $[Y_2]_2^0 \to Y[]_2^0$; $[f_{4n}^m]_1^0 \to [N_{4n+1}^{n+m}]_1^0 \; m \in \{1,...,n\}$

k) $[\bar{1}_{j1}^{**(m)}]_2^- \to [1_{in}^{**(m+1)}]_2^- \; \forall i,m \in \{1,...,n\}$; $[K_{ij}^{**(m)}]_2^- \to [K_{i(j-1)}^{**(m+1)}]_2^- \; \forall i,j \neq 1, m \in \{1,...,n\}, K \in \{1,...,n\}$; $[K_{i1}^{**(m)}]_2^- \to [K_{in}^{**(m+1)}]_2^- \; \forall i,m \in \{1,...,n\}$, $K \in \{1,...,n\}$; $[K_{i1}^{**(m)}(l)]_2^- \to [F]_2^- \; l \neq 0, K \in \{1,...,n\}, \forall i$

l) $[N_k^m]_1^0 \to [N_{k+1}^m] \; k \in \{0,...,4n\}, m \in \{1,...,2n\}$; $[N_{4n+1}^m]_1^0 \to [N_0^{m+1}]_1^0 \; m \in \{1,...,2n\}$; $[N_{4n+1}^{2n+1}]_1^0 \to []_1^-$; $[Y]_1^0 \to []_1^+$

m) $[B]_1^+ \to \texttt{yes}[]_1^+$; $[B]_1^- \to \texttt{no}[]_1^-$

NOTATION: $K_{ij}^m$ where $m = n^2 + t$ is noted also by $K_{ij}^{*(t)}$, and analogously if $m = (n^2 + (n+1)) + t$ is noted also by $K_{ij}^{**(t)}$. The objective of this notation is to make easier the comprension of the steps through the immediate recognition of the number of asterisks over which stage are the variables in (generation, row verification or column verification). On the other hand, the objects with subscript

$1j$ during the row verification stage, as well as the objects with subscript $i1$ during the column verification stage, have an invisible variable $(l)$ with $l \in \mathbb{Z}_n$ that shows which number is being checked in each moment (the number $l+1$), and that when all the checking have been done and the row or the column number takes the value $n = 0$ and we omit it. Otherwise, any number is lacking and this induces the end of the computation in such a membrane.

In summary, the rules can be divided into the groups presented in Table 1.

| **Rules** | **Stage** | **Function** |
|---|---|---|
| $(a) - (c)$ | Generation | Generate all possible completions |
| $(d) - (g)$ | Row verification | Check every symbol in every row |
| $(h) - (k)$ | Column verification | Check every symbol in every column |
| $(l) - (m)$ | Output | Synchronize and return `yes` or `no` |

**Table 1.** Summary of the role of the rule groups in the computation.

### 3.1 Summary of the computation

For the sake of simplicity, we synthesize the behavior of the system with some ideas.

Each cell from the square has an associated object $X_{i,j}$ where, if the cell contains the number $K$, $X$ takes that value. Otherwise, $X$ will keep as indeterminate. These variables are also complemented with some auxiliary variables that keep evolving in each step and are useful to differentiate the different stages of the computation. In this sense, until a certain step is not reached, certain rules cannot be applied on them. The stage division follows the usual pattern for solving **NP**-complete problems [21]:

- **Generation stage.**
- **Row checking stage.**
- **Column checking stage.**
- **Output stage.**

In the **generation stage**, for each cell of the square with an indeterminate associated object, the membrane 2 is succesively divided to assign all the possible values from 1 to $n$. At the end of this stage, all the possible extensions of the original input are available in all the membranes 2. The method to assign these $n$ values is to divide the membrane where the object $X_{ij}$ exists into two: One assigning $X = 1$ and stopping the division process in that membrane, and the other assigning $X = 1$ while allowing successive divisions to be applied. This process continues dividing by 2 adding 1 to the value of the object, one of them stopping the division and the other membrane continuing the division, until the last value that stops the process of such a variable. Instead of going through each cell, we "move" the subscripts of the objects, assigning values to the objects

whose subscript is actually 11; that is, the subscripts of the objects $X$ and $K(1 \leq K \leq n)$ are decreased by row and column in order to reach the subscript 11. When this stage is reached, the division stage of such a cell is started. This stage takes $n^3+2n^2+1$ computational steps. An initial step is taken to transform the object $g$ to $g_{11}$, and for each cell, $n$ steps for evolving the counter, 1 extra for going to the next cell and 1 more for transforming $g_{ij}^n$ to $g_{ij}^{n+1}$, making it $1 + n^2(n + 2) = n^3 + 2n^2 + 1$.

At the end of this stage, the **row checking stage** is started, where we check if the first row contains each number from 1 to $n$ exactly once. For that purpose, we check that there exists a 1 and there is no other one, and then we reduce by 1 each value from the row, repeating the process (now checking if there exists a 1 is to check if there exists a $2 = 1 + 1$). When all the values have been checked, the initial configuration is recalled and the rows are shifted (changing the subscripts of the objects) for the next row to be in the place of the first one, and the process is repeated. After repeating this process $n$ times, the whole square has been checked and the initial configuration of this stage is recalled. If every checking has been successful, then the next stage is started in such a membrane. This stage takes $n(4n + 2) = 4n^2 + 2n$ computational steps, since for each row $4n + 2$ steps are needed, $4n$ for the counter, 1 step for entering the stage and 1 step for transitioning to the next one.

The **column checking stage** is analogous to the row checking stage, but shifting the columns instead of the rows. Because of that, this stage also takes $n(4n+2) = 4n^2+2n$ computational steps. In the end, the **output stage** arrives. If any of the Latin square completions obtained during the generation stage satisfy all the checkings from the two previous stages, then an associated object to such a membrane has been evolving until sending an object $Y$ to the membrane 1, that changes the polarization to such a membrane to positive, and sends an object `yes` to the environment. If all the membranes have failed; that is, no Latin square completion fulfills all the checking, then the associated objects have stopped to evolve at some point. If there is no object $Y$ that changes the polarization of the membrane 1, an object evolves until changing to the object $N$, that changes the polarization of the membrane 1 to negative and sending an object `no` to the environment. If the answer is `yes`, the answer is returned in 5 computational steps. Otherwise, the amount of computational steps needed is $4n + 3$, $4n + 1$ steps for passing from $N_0^{2n+1}$ to $N_{4n+1}^{2n+1}$, 1 step for changing the polarization of the skin membrane and 1 step for the output.

The total number of steps that the computation takes is $n^3+10n^2+4n+6 \in O(n^3)$ in the positive case, and $n^3 + 10n^2 + 8n + 4 \in O(n^3)$ otherwise.

## 4 Correctness and computational resources

In this section, the correctness (both the soundness and completeness) of the solution is studied, as well as the amount of resources used.

### 4.1 Correctness of the construction

**Theorem 1.** *The family* $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$ *solves the Latin Square Completion problem in polynomial time.*

*Proof.* The proof follows the four stages of the computation.

During the generation stage, the objects associated with the cells of the input square are processed sequentially. The fixed values of the cells remain in the corresponding membranes, while division rules are executed in membranes corresponding where there are objects corresponding to "empty" cells for each. Thus, after all the cells have been processed, every membrane labelled by 2 encodes a total assignment of the input partial Latin square. Moreover, every possible assignment is represented by a membrane.

*Soundness:* Assume that $\Pi(n)$ outputs `yes`. By the send-out rules described, this can only occur if an object $Y$ reached the skin membrane and changed its polarization to positive. Such an object can be generated only by a membrane that completed both the row verification and the column verification.

Consider the total assignment encoded by such a membrane. During the row verification, all the symbols that encode the square are examined. For a specific row, each auxiliary object and counter verifies if there exists one of each symbol in such a row. If one symbol does not exist in the row, the rules "reject" the encoded square and the membrane does not complete this stage. Once the symbols have finished checking one row, the objects are prepared for checking the next row, until all the rows are examined. Since each row contains exactly $n$ cells whose values belong to $\{1, \ldots, n\}$, then if all the symbol appear in a row it means that each symbol appears exactly once in that row.

The verification of the columns is similar to that of the rows. Thus, the membrane finishes this stage (without "rejecting" the encoded square) if all the columns are verified to have all the symbols and, therefore, only once.

Therefore, the candidate membrane that produces $Y$ encodes a square completion where every symbol occurs exactly once in each row and exactly once in each column. Moreover, all the symbols from the original partial Latin square have been preserved throughout the generation stage, being the square completion of such a membrane an extension of the original one. Therefore, if $\Pi(n)$ outputs `yes`, the input partial Latin square admits a completion.

*Completeness:* Let us assume that the input partial Latin square $L$ admits at least one completion, and let $L'$ be one such completion. For every empty cell of $L$, the P system will generate in the first stage all the possible assignments of symbols. Since the original input partial Latin square is preserved, then the completion $L'$ is generated.

Since $L'$ is a Latin square, every row contains each symbol from $\{1, \ldots, n\}$ exactly once, and every column contains each symbol from $\{1, \ldots, n\}$ exactly once. Therefore, both the stages of row verification and column verification are "passed" by the membrane encoding such a Latin square $L'$. When these checking stages are passed, an object $Y$ is produced and sent to the skin membrane.

This object $Y$ changes the polarization of the skin membrane to positive, and the object $B$ can apply the send-out rule, sending the object `yes` to the environment. Thus, if the input partial Latin square admits a completion, then $\Pi(n)$ outputs `yes`.

### 4.2 Computational resources

The family $\mathbf{\Pi}$ is polynomially uniform, its construction is made in polynomial time with respect to the size $n$ of the input square, and the encoding can also be constructed in polynomial time. Besides, the system works in $O(n^3)$ computational steps.

The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines –that is, each member of the family $\mathbf{\Pi}$ can be built by a Turing machine in polynomial time– because for each $n \in \mathbb{N}$, the amount of resources needed to build $\Pi(n)$ is of a polynomial order in $n$:

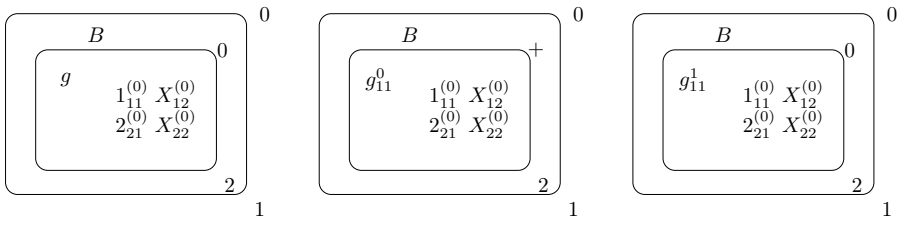
- Size of the alphabet: $5n^5 + 2n^3 + 19n^2 + 12n + 11 \in \Theta(n^5)$.
- Initial number of membranes: $2 \in \Theta(1)$
- Initial number of objects in membranes: $2 \in \Theta(1)$.
- Number of rules: $n^5 + 9n^4 + 9n^3 + 28n^2 + 14n + 9 \in \Theta(n^5)$.

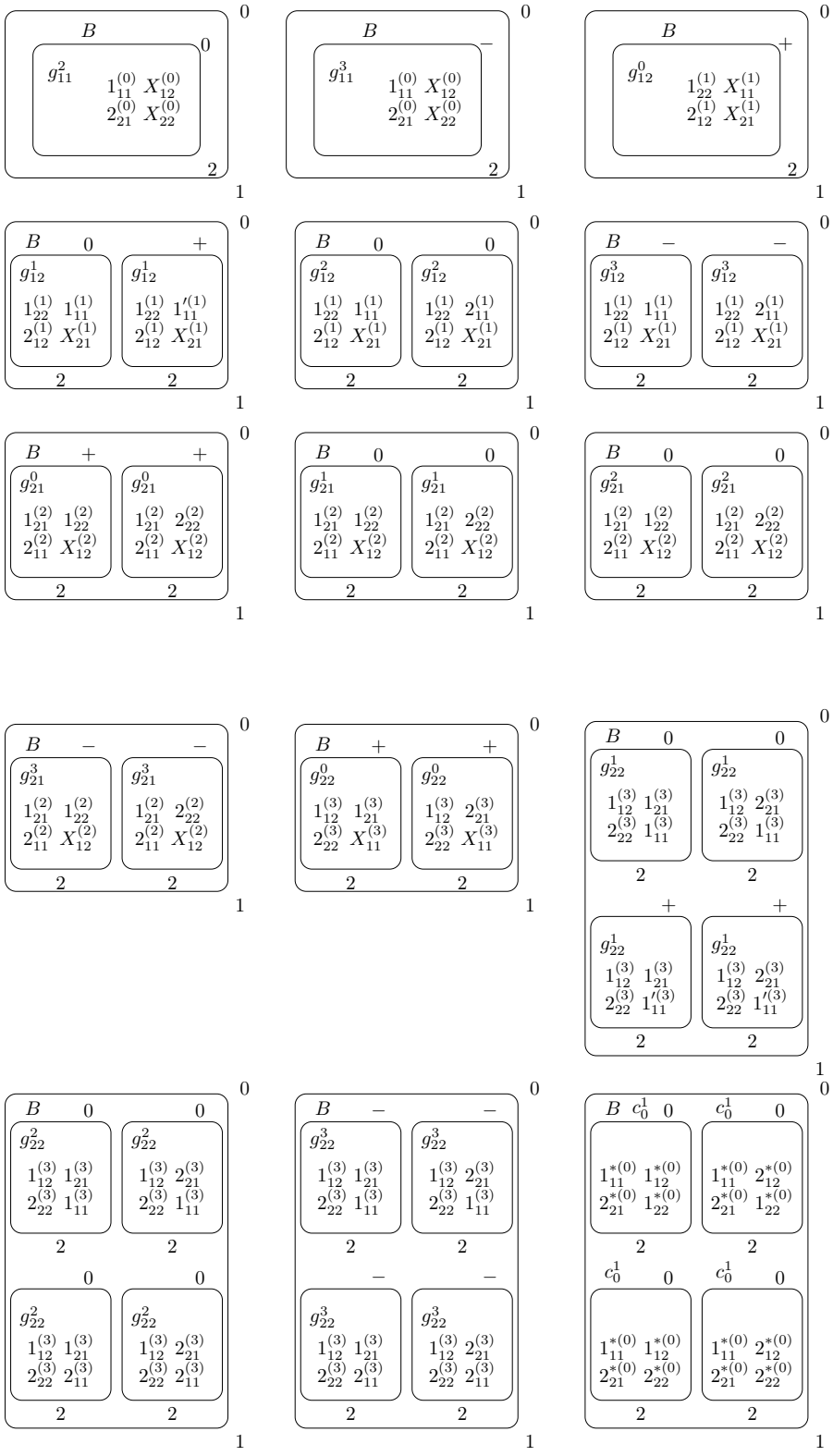
The pair $(cod, s)$ of polynomial-time computable functions defined fulfill the following: for each input partial Latin square $L$ of the `LSC` problem, $s(L)$ is a natural number, $cod(L)$ is an input multiset of the system $\Pi(s(L))$, and for each $n \in \mathbb{N}$, $s^{-1}(n)$ is a finite set.

The family is polynomially bounded, sound and complete as needed by the definitions from [18, 19].

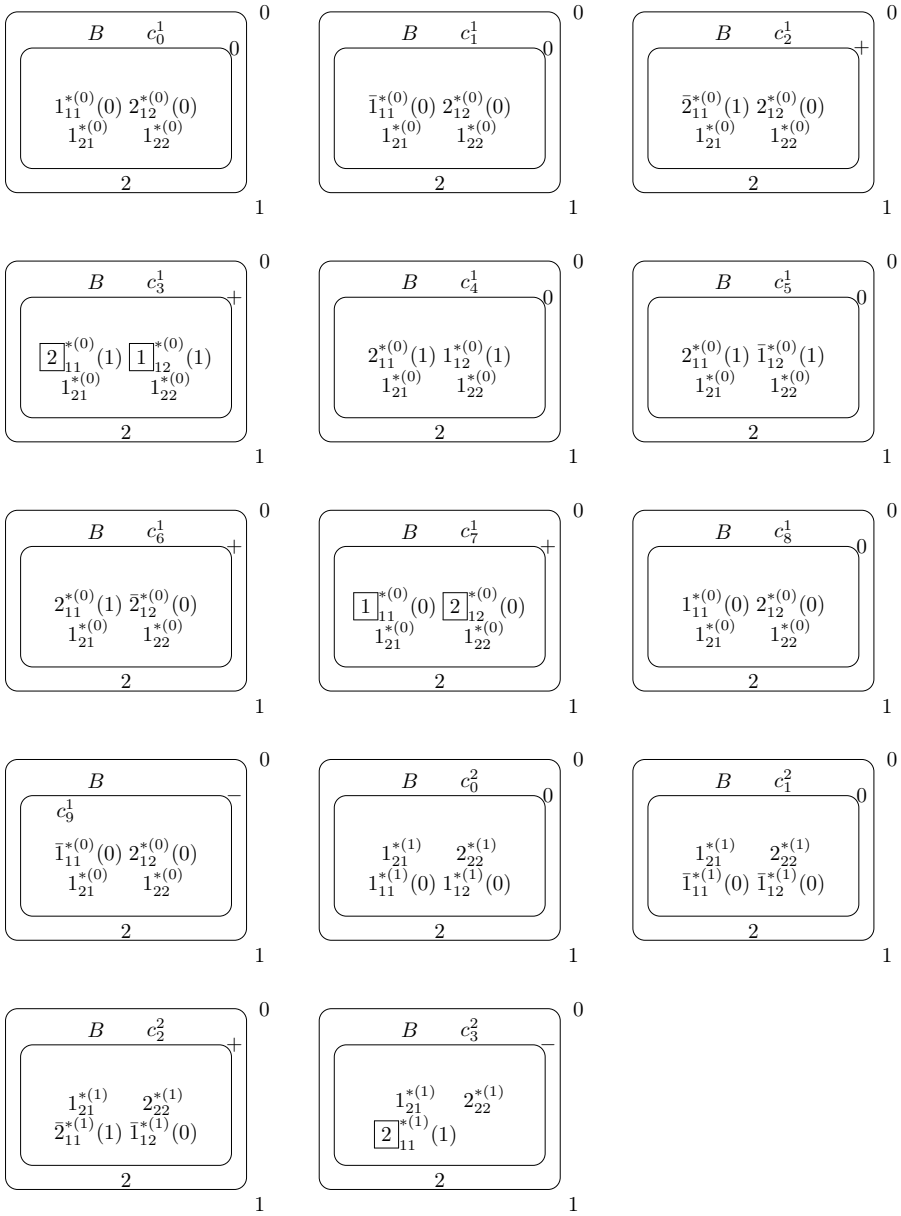
## 5 An illustrative example

For the sake of the demonstration, let the instance $L = (2, \{(1, 1, 1), (2, 1, 2)\}$. We show in this section an illustrative step-by-step solution of such a partial Latin square by using $\Pi(2)$. The following steps show the generation stage (the steps are shown left to right and top to bottom).




B
0
$g^2_{11}$
$1^{(0)}_{11}$ $X^{(0)}_{12}$
$2^{(0)}_{21}$ $X^{(0)}_{22}$
2
1
B
−
$g^3_{11}$
$1^{(0)}_{11}$ $X^{(0)}_{12}$
$2^{(0)}_{21}$ $X^{(0)}_{22}$
B
+
$g^0_{12}$
$1^{(1)}_{22}$ $X^{(1)}_{11}$
$2^{(1)}_{12}$ $X^{(1)}_{21}$
B 0 +
$g^1_{12}$
$1^{(1)}_{22}$ $1^{(1)}_{11}$
$2^{(1)}_{12}$ $X^{(1)}_{21}$
$g^1_{12}$
$1^{(1)}_{22}$ $1'^{(1)}_{11}$
$2^{(1)}_{12}$ $X^{(1)}_{21}$
B 0 0
$g^2_{12}$
$1^{(1)}_{22}$ $1^{(1)}_{11}$
$2^{(1)}_{12}$ $X^{(1)}_{21}$
$g^2_{12}$
$1^{(1)}_{22}$ $2^{(1)}_{11}$
$2^{(1)}_{12}$ $X^{(1)}_{21}$
B − −
$g^3_{12}$
$1^{(1)}_{22}$ $1^{(1)}_{11}$
$2^{(1)}_{12}$ $X^{(1)}_{21}$
$g^3_{12}$
$1^{(1)}_{22}$ $2^{(1)}_{11}$
$2^{(1)}_{12}$ $X^{(1)}_{21}$
B + +
$g^0_{21}$
$1^{(2)}_{21}$ $1^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
$g^0_{21}$
$1^{(2)}_{21}$ $2^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
B 0 0
$g^1_{21}$
$1^{(2)}_{21}$ $1^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
$g^1_{21}$
$1^{(2)}_{21}$ $2^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
B 0 0
$g^2_{21}$
$1^{(2)}_{21}$ $1^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
$g^2_{21}$
$1^{(2)}_{21}$ $2^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
B − −
$g^3_{21}$
$1^{(2)}_{21}$ $1^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
$g^3_{21}$
$1^{(2)}_{21}$ $2^{(2)}_{22}$
$2^{(2)}_{11}$ $X^{(2)}_{12}$
B + +
$g^0_{22}$
$1^{(3)}_{12}$ $1^{(3)}_{21}$
$2^{(3)}_{22}$ $X^{(3)}_{11}$
$g^0_{22}$
$1^{(3)}_{12}$ $2^{(3)}_{21}$
$2^{(3)}_{22}$ $X^{(3)}_{11}$
B 0 0
$g^1_{22}$
$1^{(3)}_{12}$ $1^{(3)}_{21}$
$2^{(3)}_{22}$ $1^{(3)}_{11}$
$g^1_{22}$
$1^{(3)}_{12}$ $2^{(3)}_{21}$
$2^{(3)}_{22}$ $1^{(3)}_{11}$
+ +
$g^1_{22}$
$1^{(3)}_{12}$ $1^{(3)}_{21}$
$2^{(3)}_{22}$ $1'^{(3)}_{11}$
$g^1_{22}$
$1^{(3)}_{12}$ $2^{(3)}_{21}$
$2^{(3)}_{22}$ $1'^{(3)}_{11}$
B 0 0
$g^2_{22}$
$1^{(3)}_{12}$ $1^{(3)}_{21}$
$2^{(3)}_{22}$ $1^{(3)}_{11}$
$g^2_{22}$
$1^{(3)}_{12}$ $2^{(3)}_{21}$
$2^{(3)}_{22}$ $1^{(3)}_{11}$
0 0
$g^2_{22}$
$1^{(3)}_{12}$ $1^{(3)}_{21}$
$2^{(3)}_{22}$ $2^{(3)}_{11}$
$g^2_{22}$
$1^{(3)}_{12}$ $2^{(3)}_{21}$
$2^{(3)}_{22}$ $2^{(3)}_{11}$
B − −
$g^3_{22}$
$1^{(3)}_{12}$ $1^{(3)}_{21}$
$2^{(3)}_{22}$ $1^{(3)}_{11}$
$g^3_{22}$
$1^{(3)}_{12}$ $2^{(3)}_{21}$
$2^{(3)}_{22}$ $1^{(3)}_{11}$
− −
$g^3_{22}$
$1^{(3)}_{12}$ $1^{(3)}_{21}$
$2^{(3)}_{22}$ $2^{(3)}_{11}$
$g^3_{22}$
$1^{(3)}_{12}$ $2^{(3)}_{21}$
$2^{(3)}_{22}$ $2^{(3)}_{11}$
B $c^1_0$ 0 $c^1_0$ 0
$1^{*(0)}_{11}$ $1^{*(0)}_{12}$
$2^{*(0)}_{21}$ $1^{*(0)}_{22}$
$1^{*(0)}_{11}$ $2^{*(0)}_{12}$
$2^{*(0)}_{21}$ $1^{*(0)}_{22}$
$c^1_0$ 0 $c^1_0$ 0
$1^{*(0)}_{11}$ $1^{*(0)}_{12}$
$2^{*(0)}_{21}$ $2^{*(0)}_{22}$
$1^{*(0)}_{11}$ $2^{*(0)}_{12}$
$2^{*(0)}_{21}$ $2^{*(0)}_{22}$

Then, the generation stage is over, and it can be seen that there exists a membrane 2 for each possible completion of $L$. It is easy to see that only one of them (the one at the top right) corresponds to a solved Latin square. To show the checking stage, we restrict the example to the first part, the row validation (since column validation is analogous) and the case of a single membrane after the generation stage. More precisely, we compute a complete but not valid instance: $L = (2, \{(1,1,1),(1,2,2),(2,1,1),(2,2,1)\}$



The polarization of the membrane has changed to negative in the final step, and there are no objects either in the membrane or outside the membrane that can change the electrical charge. Only rules from $(g)$ and $(k)$ are applicable, corresponding to the $n$ rows and $n$ columns of the Latin square, depending on where the candidate has failed. Within such a membrane, after at most $2n$ additional steps, no further rule will be able to be applied. The associated object $c_k^2$ evolves until $c_8^2$, and since it is not able to be sent into the membrane, it will be transformed into a variable $N_9^2$, that will eventually change the polarization of the membrane 1 to negative. If there are membranes that contain a correct Latin square completion this configuration will not be achieved since other objects would change the polarization of the membrane 1 to positive.

## 6 Conclusions and future work

In this work, an explicit solution to the Latin square completion problem has been provided by means of recognizer P systems with active membranes. The idea follows the traditional pattern where there are a generation stage, a checking stage and an output stage. In this case, in the first one, all the possible completions (even wrong ones) are generated thanks to the use of division rules, that generate an exponential number of membranes in polynomial time. Then, the checking is divided into two verification stages: Rows verification and columns verification. Once it is finished, the system outputs an object `yes` if and only if there exists a correct Latin square completion for the original partial Latin square. Even when the recognizer P systems return the `yes` and `no` objects, the membranes labelled by 2 that generated an object $Y$ are supposed to have a correct Latin square completion, thus marking them in some sense (or removing the others) would make it easy to find a correct answer and not only the solution for the decision problem.

Although the proposed construction works in polynomial time and has a polynomial description, its direct implementation may still be demanding (both the working alphabet and the number of rules grow as $\Theta(n^5)$, and the generation stage produces an exponential number of membranes). Therefore, the construction should be understood as a theoretical solution. From an implementation perspective, more compact encodings and reductions in the number of auxiliary objects and rules could improve its scalability.

An interesting research line is to use computing P systems instead of recognizer P systems in order to directly send out the encoding of a correct answer to the environment, to treat the system as a "black box" that returns the solution to the provided input. Another possible direction is to incorporate pruning mechanisms into the generation stage in order to avoid producing candidate completions that are already known to be invalid. This could reduce the number of generated membranes and avoid unnecessary verification steps for some instances, although the worst-case asymptotic complexity of the solution would remain unchanged. Besides, it would be interesting to lower the number of ingredients (i.e. remove the polarizations) of the solution. Another interesting open

research line is to extend this type of method to other variants such as diagonal squares, Graeco-Latin squares or with additional restrictions, such as Sudoku, apart from other interesting puzzles.

**Acknowledgments.** This work is part of Project PID2025-174204NB-I00, funded by the Ministry of Science, Innovation and Universities (MICIU), the State Research Agency (AEI), DOI 10.13039/501100011033, and the European Regional Development Fund (ERDF), European Union.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Colbourn, C.J.: The complexity of completing partial Latin squares. Discrete Applied Mathematics **8**(1), 25–30 (1984). https://doi.org/10.1016/0166-218X(84)90075-1
2. Colbourn, C.J., Dinitz, J.H. (eds.): Handbook of Combinatorial Designs. Chapman & Hall/CRC, Boca Raton, 2 edn. (2006). https://doi.org/10.1201/9781420010541
3. Deodhare, D., Sonone, S., Gupta, A.: A generic membrane computing-based Sudoku solver. In: 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT). pp. 89–99. IEEE (2014). https://doi.org/10.1109/ICICICT.2014.6781258
4. Díaz-Pernil, D., Fernández-Márquez, C.M., García-Quismondo, M., Gutiérrez-Naranjo, M.A., Martínez-del Amor, M.A.: Solving Sudoku with membrane computing. In: 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA). pp. 610–615. IEEE (2010). https://doi.org/10.1109/BICTA.2010.5645195
5. Easton, T., Parker, R.G.: On completing Latin squares. Discrete Applied Mathematics **113**(2–3), 167–181 (2001). https://doi.org/10.1016/S0166-218X(00)00282-1
6. Goudet, O., Hao, J.K.: A massively parallel evolutionary algorithm for the partial Latin square extension problem. Computers & Operations Research **158**, 106284 (2023). https://doi.org/10.1016/j.cor.2023.106284
7. Gutiérrez-Naranjo, M.A., Martínez-del Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Solving the $N$-queens puzzle with P systems. In: Seventh Brainstorming Week on Membrane Computing. vol. 1, pp. 199–210. Fénix Editora, Sevilla (2009)
8. Jin, Y., Hao, J.K.: Solving the Latin square completion problem by memetic graph coloring. IEEE Transactions on Evolutionary Computation **23**(6), 1015–1028 (2019). https://doi.org/10.1109/TEVC.2019.2899053
9. Keedwell, A.D., Dénes, J.: Latin Squares and Their Applications. North-Holland, Amsterdam, 2 edn. (2015)
10. Pan, S., Wang, Y., Yin, M.: A fast local search algorithm for the Latin square completion problem. In: Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence. vol. 36, pp. 10327–10335. AAAI Press (2022). https://doi.org/10.1609/aaai.v36i9.21274
11. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693

12. Păun, G.: P systems with active membranes: Attacking NP-complete problems. Journal of Automata, Languages and Combinatorics **6**(1), 75–90 (2001)
13. Păun, G.: Membrane Computing: An Introduction. Natural Computing Series, Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/978-3-642-56196-2
14. Păun, G., Rozenberg, G.: A guide to membrane computing. Theoretical Computer Science **287**(1), 73–100 (2002). https://doi.org/10.1016/S0304-3975(02)00136-6
15. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford (2010). https://doi.org/10.1093/oso/9780199556670.001.0001
16. Păun, G., Suzuki, Y., Tanaka, H., Yokomori, T.: On the power of membrane division in P systems. Theoretical Computer Science **324**(1), 61–85 (2004). https://doi.org/10.1016/j.tcs.2004.03.053
17. Pérez-Jiménez, M.J., Riscos-Núñez, A.: Solving the subset-sum problem by P systems with active membranes. New Generation Computing **23**(4), 339–356 (2005). https://doi.org/10.1007/BF03037637
18. Pérez-Jiménez, M.J., Romero-Jiménez, Á., Sancho-Caparrini, F.: Complexity classes in models of cellular computing with membranes. Natural Computing **2**(3), 265–285 (2003). https://doi.org/10.1023/A:1025449224520
19. Pérez-Jiménez, M.J., Romero-Jiménez, Á., Sancho-Caparrini, F.: Computationally hard problems addressed through P systems. In: Ciobanu, G., Păun, G., Pérez-Jiménez, M.J. (eds.) Applications of Membrane Computing, pp. 315–346. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/3-540-29937-8_12
20. Pérez-Jiménez, M.J., Romero-Jiménez, Á., Sancho-Caparrini, F.: A polynomial complexity class in P systems using membrane division. Journal of Automata, Languages and Combinatorics **11**(4), 423–434 (2006)
21. Romero-Jiménez, Á., Orellana-Martín, D.: Design patterns for efficient solutions to NP-complete problems in membrane computing. In: Enjoying Natural Computing, Lecture Notes in Computer Science, vol. 11270, pp. 237–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00265-7_19
22. Salomaa, A.: Formal Languages. ACM Monograph Series, Academic Press, New York (1973)
23. Singh, G., Deep, K.: A new membrane algorithm using the rules of particle swarm optimization incorporated within the framework of cell-like P-systems to solve Sudoku. Applied Soft Computing **45**, 27–39 (2016). https://doi.org/10.1016/j.asoc.2016.03.020
24. Zandron, C., Ferretti, C., Mauri, G.: Solving NP-complete problems using P systems with active membranes. In: Unconventional Models of Computation, UMC 2000. pp. 289–301. Springer, London (2001). https://doi.org/10.1007/978-1-4471-0313-4_21

# A Visual Discrete Event-based Simulator for the Optimization of Plant Defense Strategies against Herbivores as Application of a Tissue P System

Lucas Dietrich[1], Benjamin Förster[2], Peter Langendörfer[2], and Thomas Hinze[3]

[1] Hamburg University of Technology, Computer Center RZ,
Am Schwarzenberg-Campus 3E, D-21073 Hamburg, Germany
lucas.dietrich@tuhh.de

[2] Brandenburg University of Technology Cottbus-Senftenberg, Institute of Computer Science, Chair of Wireless Systems, Postfach 10 13 44, D-03013 Cottbus, Germany
{benjamin.foerster,peter.langendoerfer}@b-tu.de

[3] Friedrich Schiller University Jena, Department of Bioinformatics,
Inselplatz 5, D-07743 Jena, Germany
thomas.hinze@uni-jena.de

**Abstract.** Plants come with sophisticated strategies to survive within a highly competing environment. In addition, they need to resist frequent attacks from a variety of herbivores acting alone, in small groups, or in swarms. Since the amount of energy a plant might invest in defense and reproduction is limited, a complex optimization problem emerges. In a shared habitat, plants fight herbivores by shape and camouflage, by the release of specific toxins, or by attracting predators of herbivores. Furthermore, plants alert their surrounding field by signaling compounds in the event of an assault. Transported by air or through a network of roots, signaling compounds reach neighbors to trigger their defense. The offspring of a plant commonly grow within a certain distance to benefit from symbiotic protection. We introduce a grid-based visual simulation software for detailed configuration and subsequent processing of the behavior of the resulting system in time and space. The underlying formalization utilizes an extended form of tissue P systems with environment. In terms of solution to a computational optimization problem inspired by nature, settings with low energy need and long life able to cope with different patterns of attack can be figured out and analyzed. Applications include novel techniques for efficient construction and secure operation of sensor networks.

## 1 Introduction

Plants have created a fascinating variety of mechanisms to protect themselves and their conspecifics or symbiotic partners against manifold herbivorous insects, predatory animals, and microbial pathogens [5]. Since most plants reside in a fixed position on earth with roots directed toward the ground and aerial leaves harvesting sunlight, they cannot move as a whole in order to escape. Typically,

their offspring settle within the closest environment, but keeping a certain distance to each other. It should be noted that plants of the same species tend to form networks of individuals spread over a region. They share the ground with other plants, trees, and mushrooms in a mixed neighborhood. The resulting habitat acts in a highly competitive manner with symbiotic elements emerged to cope with lack of resources, nutrients, and individual space [28].

The optimal placement of their offspring in the surrounding landscape turns out to be merely one challenge for a plant or a family of plants that aims to survive and reproduce. A much greater threat is caused by a variety of *herbivores*. They preferably feed on leaves but also on flowers, or other components of plants for nutrition. Herbivores exist in many forms. Some of them mostly act alone like deer or giraffes. Others favor small groups such as wild pigs or bunnies. In particular, swarms consisting of hundreds or thousands of individuals can be destructive in a disastrous way. Insects, worms, ants, and other articulate animals have the ability to damage wide areas in short time up to the extinction of plant species.

In general, plants have been equipped with a plethora of different and fine-tuned methods and techniques for protection against herbivores [9]. The genomes found in the flora belong to the largest among all life forms. Although there is no central nervous system or brain, a plant typically realizes a feeding attack since the destroyed cell structures at the edge of a bite mark become contaminated. The presence of foreign molecules left by the herbivorous animal initiates a biochemical signaling cascade which in turn can be kept locally or - in case of a larger attack - incorporates most parts of the plant to get activated. The amount, spatial distribution, and combination of foreign molecules reveal information about the enemies and identify the type of attacker along with the intensity and directions of feed. Now, signaled cells around the locations of damage can start a specific defense.

Defense and deterrence consume *energy*, a very limited resource. Young and small plants with low *leaf area* in total might harvest much less energy from sunlight than full-grown trees with approximately $1,200\text{m}^2$ on average up to a maximum of around $15,000\text{m}^2$ [10]. Root ends mainly contribute water and nutrients, but rarely exploitable energy. Metabolism and life support require an individual threshold of permanent energy need for existence. Energy resources beyond the threshold are available for reproduction and protection.

The behavior of plants in terms of energy use reflects the process of solving a complex *optimization problem* about how to survive and reproduce with a minimal amount of energy fending off all relevant herbivores. In some cases, it seems to be advantageous to spend energy for the permanent development of prickles, thorns, or leaves with saw-toothed shape [11]. In this way, single individuals of larger animals might lose their willingness to feed. The same holds for the release of bitter substances which in turn need to be produced by the plant consuming energy. Ideally, the benefit consists in the generation of a memory effect, and herbivorous animals remember their experience and avoid the plant for long time. In contrast, this strategy fails at swarm attacks. Here, the plant

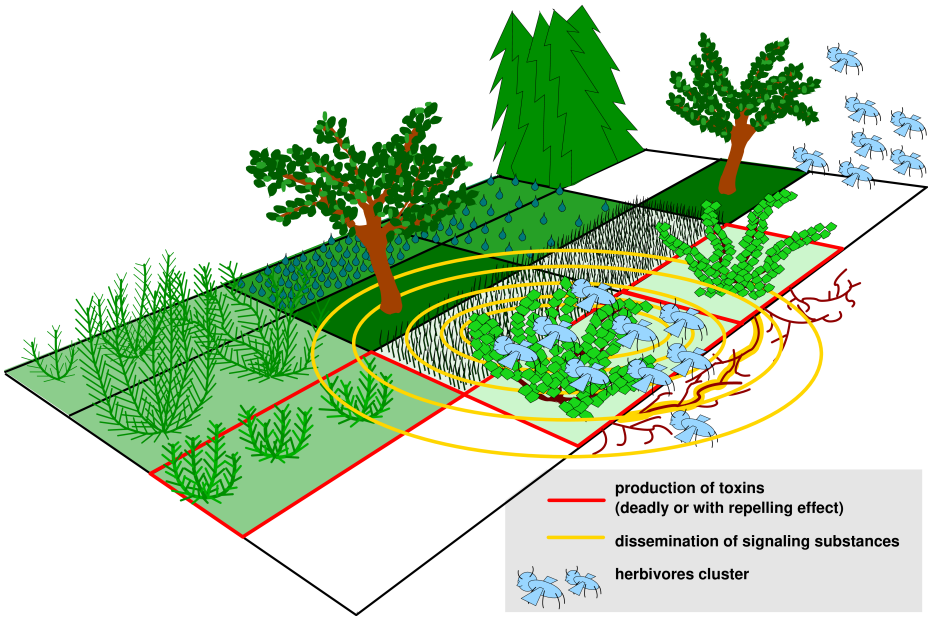



Fig. 1: Grid scheme with spatial distribution of plants, herbivores clusters, dissemination of signaling compounds, and production of toxins.

must invest much more energy in the rapid production of effective specific toxins or drugs. While toxins aim at a deadly effect that kills the individual attacker, drugs diminish the alertness and awareness of attackers in combination with attracting their enemies. Moreover, families of plants can alert each other. If one plant is attacked, it sends signaling compounds to other members of its family. This can happen by spreading over air, but also through a mycorrhiza, an underground symbiotic association between roots and a fungus. The alerted plants then start to prepare for defense on their own, see Figure 1. So, a swarm can be eliminated in a short time, but with a high use of energy. Especially families of plants are capable of regeneration after some members have died off. The clue might be found in an optimal spatial distribution of the plants in the landscape. The resulting topology of the vegetation cover in total gives a solution to the optimization problem mentioned above. In addition, the response scheme of individual plants against herbivores marks a part of the optimal solution.

What stands out is the idea of a software simulator that enables a fine-grained configuration of a scenario consisting of all players like families of plants, clusters of herbivores, and a pool of signaling molecules, bitter substances, and toxins at an abstract level captured by appropriate parameters and attributes. The initial distribution of plants and herbivores is done on a grid in a freely configurable manner. After the whole setting had been specified, the simulation runs. Its progression is controlled by discrete time steps and events. Typical events are for

instance the update of the energy level of an individual plant, the elimination of a plant whose energy level fell under the predefined threshold for survival, placement of the offspring of a plant, loss of herbivore individuals, motion of a cluster of herbivores on the grid, and many others. Performing an event might imply the generation of additional events at later time points. In conjunction with simulation, the behavior of the entire system becomes visible and can be analyzed by the software on the fly. This feature allows for comparison of different initial settings of systems with respect to energy need, reproduction rate, and overall fitness. In the next stage, an optimization towards robust and persistent families of plants with low energy need is feasible. To the best of our knowledge, this is the first attempt to provide such a tool adopting natural computation in a moderately abstract processing scheme for the protection of plants against herbivores. Our software named *PlantProtectionSim* is freely available from Github[4].

The security of *wireless sensor networks* (WSNs) can be informed by strategies observed in plant defenses, as both systems are resource-constrained and must operate reliably in hostile environments [15]. In such a situation, hundreds to thousands of sensor nodes are deployed to carry out monitoring or controlling tasks like wildfire detection, precision agriculture (for example, irrigation control), or infrastructure monitoring (for example, bridges). Like plants, WSNs cannot maintain absolute protection and must balance the energy expenditure of defense with long-term operational needs. Plants combine constitutive and inducible mechanisms and even signal neighboring organisms when under attack [14]. This biological analogy motivates the design of distributed and collaborative security schemes for WSNs, aiming to broaden threat coverage while limiting the burden on individual nodes.

The paper is structured as follows. Section 2 sketches the biological background and reflects related work. In Sections 3 and 4, we describe the underlying model, its representation by an extended tissue P system [1, 20, 23], and its implementation for the simulator software, while Section 5 is dedicated to a user manual in brief. A case study will be addressed in Section 6. Eventually, Section 7 discusses the results, draws conclusions, and sheds light on future work.

## 2 Biological Background of Plant Protection

Plant protection strategies against potential attackers can be classified into *constitutive*, *induced*, and *activated* defense mechanisms from a systems biology perspective [7, 26]. All of them come with specific advantages worth to be generalized towards technical counterparts.

The simplest way for small plants to survive consists of getting *invisible* if necessary. Grass follows this strategy, for example. It can grow rather fast, stores nutrients in its roots, and it is able to survive for weeks without any contact with sunlight. After a herbivore has fed off the halms, the grass regenerates beneath

[4] https://github.com/lucas-diet/PlantProtectionSim

the ground. This behavioral strategy does not require effort or resources for a dedicated defense or alert. For the long-term survival of grass in the savanna, this seems to be successful, but grass is restricted in its size and in its ability to persist in the relative darkness of forests. When plants started to compete with each other for the maximal portion of sunlight, they needed to become larger and more durable, which implies more sophisticated strategies against attackers. If a plant is under potential threat through an attacker, it makes use to evaluate whether a countermeasure will be expedient, since a reaction also causes costs and stress to it [24].

In case a plant does not actively react, there is a *constitutive* defense mechanism in place that protects the plant against a multitude of potential threats without additional costs [27]. Examples of constitutive defense mechanisms are spikes or tooth-shaped outer forms of leaves grown by nature in the plant. Complementary, some plants continuously accumulate poisonous chemicals in their leaves or stipes. Here, chemicals commonly arise as side products of the underlying metabolism. A typical example is tobacco, which produces nicotine in this way, a mild poison that affects the nervous system of herbivores. A brindle coloring of the leaves helps a couple of plants in camouflage. All constitutive defense mechanisms have in common that no energy-consuming signaling is necessary inside and among plants. Generally, the constitutive defense mechanism makes herbivores avoid the corresponding plant and exemplars of the same species for a certain time. Rarely does this strategy lead to the death of herbivores. Additional energy need for the plant remains low, but its genome requires extension.

A more complex defense pattern is defined by *induced* security mechanisms whose main feature lies in a high precision against the specific attacker in combination with an outstanding flexibility to cope with different types of enemies [21]. The behavioral pattern for defense gets induced upon an ongoing attack or in case an attack is signaled by other surrounding plants of the same species. Plants react in a purposive manner. The resulting cascade of biochemical response comprises successive steps of action and interaction. Typically, the processing scheme starts with a form of *signaling* [4]. Herbivores destroy the inner structure of leaves or stipes, which in turn implies one or more wounds. In addition, they leave indications about their identity by contamination of the wounds. The first stage of signaling releases messenger substances that circulate within the affected plant. The presence of those messengers initiates a dedicated biochemical production of response molecules. They in turn initiate a *chemical defense* by opening reservoirs of highly concentrated poisons or chemicals that cause a bad taste directly transported to the wounds [22]. Alternatively, direct defense can attract herbivore enemies using pheromones or attractants. A third type of defense revokes the chlorophyll of their leaves, which stops photosynthesis, but also eliminates nutrients from their leaves. This comes along with a temporary change in the colors of leaves subsumed by the term *aposematic signaling* [19]. Furthermore, the plant starts to inform its neighbors about the ongoing attack. The corresponding second stage of signaling acts in terms of an alert.

For this purpose, the corresponding signals exhibit a biochemical nature [16]. Although some gaseous chemical signals such as ethylene can be communicated over air, a variety of protein-based signals are exclusively communicated via roots of plants and fungi and often through the symbiotic mycorrhiza [6]. Typically, these messaging substances make use of electrical forces for transport. An alert signaled from plants in its neighborhood has the same effect as a direct attack. After a plant has received an alert and the concentration of involved signaling molecules exceeds a certain threshold, the chemical defense or aposematic signaling is set into operation. Induced security mechanisms run for a certain amount of time after an attack occurs or has been signaled. Then, they stop and turn into regeneration of the plant.

Induced defense mechanisms turn out to be precise and productive [12]. Herbivores are usually devitalized or even eliminated. Some animals learn to circumvent the effect of toxins, especially in the case of signaling over air. They simply feed against the wind direction. The plant must have resources to exploit a large amount of energy in a short time to produce all the required substances by means of the corresponding genes.

*Activated* defense mechanisms complete the variety of strategies found in plants [8]. Once triggered, those mechanisms remain permanently active until the plant dies. An example is an unstable toxin generation in varying leaves with random intensity over time. After the production of those toxins has been triggered, the plant will produce them sporadically in irregular periodicity and at different locations in the plant. As soon as this defense mechanism is activated, it is no longer reversible and consumes energy continuously. For scenarios in which waves of attacks appear in an iterative manner, activated defense mechanisms save time and enable a much faster response in spite of high energy costs.

Each of the chosen concepts has an impact on the fitness of the plant. Therefore, it limits the frequency and complexity of the reactions that a plant can impose. The goals of different defense mechanisms are either to stop an attack or limit the caused damage to the current target or future targets represented by nearby plants.

The chemical signaling process is the primary process for informing other plants about ongoing attacks in their environment. The most common inter-plant signaling concept with the largest reach is the chemical signaling process [25]. Plants produce a multitude of messenger molecules. Those can spread through air to warn neighboring plants about threat scenarios [13]. Different types of molecules impose different fitness costs to the plant. In addition, they have different molecular mass and therefore the distances vary, and these molecules can bridge and inform other plants about different threat scenarios. Moreover, it has to be noticed that for particular types of chemical signals, there is a corresponding specific subset of plants that understand them. For this reason, plants might signal each other hierarchically.

Hence, a plant produces molecule $A$ as signals which is understood by a subset of surrounding plants, which in turn start producing the chemical signal $B$ alerting a different subset of plants. The result is a dynamic mixture of chemicals

in the air that always surround those plants. Each plant has a concept of biological signal processing implemented on a cellular level [2]. With that, the cells are able to distinguish and evaluate the different signals. Furthermore, plants decide about their behavior pattern based on activation functions and biases about whether and how to react to potential threats [9].

The long-term process of mutual interaction between plants and herbivores results in complex behavioral patterns along with the optimal spatial distribution of plants on the ground and optimal exploitation by herbivores, respectively. Having a plethora of biological data at hand, a deeper understanding of the interplay of defense mechanisms acting as a whole becomes feasible. To do so, the first step is to develop simulation software that integrates all relevant biological aspects in an appropriate level of abstraction.

## 3 Description of the Simulator from a Use Case and Feature Perspective

The user can configure scenarios within the simulation system in a relatively flexible manner, combining plants, their spatial distribution and interactions, as well as available protective mechanisms, together with herbivores and their respective abilities and traits. In certain respects, the simulation system resembles a computer game in which a set of actors is first defined and configured, after which these actors encounter one another and interact dynamically. We arrange the software to mimic a visual discrete event-based simulator due to its ability to cope with rapid local changes.

### 3.1 Plants

As a representation of the flat landscape, a quadratic grid of up to 80 by 80 unit squares is employed, hereafter referred to as *grid cells*. Each cell may be occupied by no more than a single plant. The user specifies the number of *plant species* to be distinguished (at least one, at most sixteen). Each species is assigned a distinct color. A plant is placed on a cell by marking that cell with the corresponding color.

Each plant can be initialized with a user-defined number of *energy* units. Within the same species, this initial value may vary, for instance due to differences in plant height, age, or total leaf area. In addition, the system is configured to determine the increase of plant energy per discrete simulation step. For each individual plant, the user also specifies a minimum threshold of energy units required for survival. If, during the simulation, a plant's energy level falls below this threshold, the plant is removed from the grid.

Plants are capable of reproduction over time. The user may specify whether, and after how many simulation steps, an individual plant produces a defined number of offspring of the same species. Each offspring is again initialized with a user-defined number of energy units and is randomly placed in a free grid cell at a distance and direction from the parent plant, both determined within a

specified minimum–maximum range. If no suitable free cell is available under these conditions, the offspring is not generated.

Throughout the simulation, each cell occupied by a plant displays the percentage of energy the plant currently possesses relative to its initial energy value. This percentage may exceed or fall below 100%. Plants located in immediately adjacent grid cells can form pairwise symbiotic connections, even if they belong to different species. The user initiates such a connection by right-clicking on one of the involved grid cells. Using a popup window, each corresponding adjacent cell needs to be selected to establish the connection, which is then visualized by a black line. Through these connections, signaling compounds can propagate during the course of the simulation.

### 3.2 Herbivores

The configuration of herbivores likewise begins with the specification of how many *herbivore species* are to be included (0 to 15). Herbivores may act either in swarms or individually. To represent this, we introduce the concept of a *herbivores cluster*. A herbivores cluster consists of a freely configurable initial number of individuals (at least one). All individuals within a cluster belong to the same herbivore species and always occupy the same grid cell simultaneously. Each defined herbivores cluster can be freely placed by the user on an initial grid cell, regardless of whether the cell is already occupied by a plant and independent of the plant species. Visually, clusters are represented by a dot symbol whose size depends on herbivore type. Multiple herbivores clusters may occupy the same grid cell simultaneously.

Herbivores clusters located on grid cells without plants move toward the nearest plant. If several plants are equally close, one is chosen at random. For each herbivores cluster, the user specifies how many simulation steps are required for the cluster to move into an adjacent grid cell. Herbivore movement follows an idealized straight line across the grid and is discretized in time by mapping it to the corresponding sequence of grid cells intersected by that line.

### 3.3 Substances

As a further element of the system, *substances*, namely *signaling compounds* and *toxins*, are configured. Between 0 and 15 distinct substances may be defined. Each substance can be assigned an individual name, and the user must specify whether it is a signaling compound or a toxin.

The configuration of interactions between plants, herbivores clusters, and substances is carried out using interaction matrices (see Section 5), which the user defines prior to the simulation.

For each signaling compound, it must be specified which plant species are capable of emitting it and which species are receptive to it. Additionally, it must be defined which combinations of herbivore species trigger the emission, and what minimum number of individuals per herbivores cluster is required for this trigger. Each signaling compound can be set to propagate either through the air

or via grid-cell connections. Airborne spread occurs in concentric circles around the emitting plant. A parameter specifies after how many time steps the radius of this circle expands by one additional grid-cell length. Compounds spreading via grid-cell connections require a parameter indicating after how many time steps adjacent connected cells are reached.

The production of signaling compounds consumes plant energy. Accordingly, each compound is assigned a configurable cost, expressed as the number of energy units deducted from the plant's energy per time step. A single plant may emit multiple signaling compounds simultaneously. The presence of signaling compounds is visualized by patterns in the affected grid cells. Production ceases once the plant is no longer attacked by the herbivore species that originally triggered emission. For each compound, the user defines how many additional time steps it remains active after production stops. Once this period expires, the entire effective area originating from the triggering plant is removed, though overlapping active areas generated by other plants of the same compound remain unaffected.

The properties and interactions of toxins are specified in an analogous way. For each toxin, the producing plant species are selected and the energy cost per time step is defined. An interaction matrix determines which combinations of signaling compounds and/or herbivore species, together with the minimum herbivores cluster size, trigger toxin production, as well as how many time steps must pass before the toxin takes effect on the plant. Toxins are assumed either to kill herbivore individuals or to repel herbivores clusters. The user specifies for each toxin which herbivore species are targeted and, in the lethal case, how many individuals per cluster are eliminated per time step. In the repellent case, the producing plant is removed from the perception of affected herbivores clusters, causing them to move toward the next nearest plant without individual losses. The presence of toxins is visualized in the grid cell of the producing plant, using distinct color patterns. Toxin production ceases once the plant is no longer attacked by the triggering herbivores. For simplicity, toxins are assumed to have no residual effect: once production ends, their influence disappears immediately, leaving the plant both free of the toxin and once again a potential target for herbivores clusters.

To complete the interaction configuration, the feeding success of herbivores clusters is specified. For each herbivore species, the number of energy units derived from consumed plants that is required to generate an additional individual within the cluster is defined. As soon as a herbivores cluster succeeded in doubling its number of individuals compared to its initial value, the herbivores cluster becomes separated into two equal independent herbivores clusters.

### 3.4 Global Parameters

Finally, parameters for the simulation run itself are set. The user specifies the duration of a simulation step and chooses among several options for determining the end of the run: At least one of the following conditions needs to be set: (1) predefined number of simulation time steps is reached (2) extinction of a

specific plant species, (3) extinction of all plant species, (4) extinction of a specific herbivore species, (5) extinction of all herbivore species, (6) reaching a defined threshold for the total plant energy, or (7) reaching a defined threshold for the total number of herbivore individuals.

## 4 Modeling using Tissue P System with Rules Controlled by Timers for Conditional Events

When thinking about promising candidates for formal modeling prior to simulator software implementation, we strive for a suitable membrane system. Within the universe of corresponding P systems, the framework of *tissue P systems* gives the best fit and comes with numerous features. We follow the original approach of tissue P systems introduced in [20]. Since we prefer a direct modeling strategy from the plant protection scheme and its behavioral patterns into dedicated systems components and rules without extensive transformations based on a Turing-complete minimalistic descriptive model, there is a need for reasonable extensions. Two aspects have been considered in this context: The handling of *energy* within each membrane of a tissue P system is addressed in [1] while *states* in membranes, a prerequisite for conditional rules taking into account the overall configuration of a membrane one time step ago, have been explored in [23]. Beyond energy and inner states of membranes, our modeling approach needs to cope with longer time spans between cause and resulting effect, for instance the production period of toxins and signaling compounds consuming much more than one time step. To this end, we will equip rules with *timers* in order to delay conditional application. This new feature provides the desired flexibility for direct modeling. Application of rules follows the notion of a time-discrete series of events created along with execution of rules. In case, several rules make use of the same point in time, application becomes serialized by random.

In order to describe protection of plants against herbivores, we define a *tissue P system with rules controlled by timers* as a construct of the form

$$\Pi = (O, V_{n,n}, E, \mathcal{R}, t_{\text{fin}}), \tag{1}$$

where

- $O = \{e, s_1, \ldots, s_{15}, b_1, \ldots, b_{15}, l_1, \ldots, l_{15}, h_{1,\tau}, \ldots, h_{15,\tau}\}$
  is the alphabet of objects. Here, we symbolize
  - elementary portion of energy $e$ (enriched within a plant)
  - signaling compounds $s_i$
  - bitter substances $b_j$ with repelling effect on herbivores
  - toxins $l_k$ with lethal effect on herbivores
  - herbivore species $h_{m,\tau}$ present in cluster generation $\tau \in \mathbb{N}$
- $V_{n,n}$ with grid size $n \in \{1, \ldots, 80\}$ is a quadratic matrix written as a set of grid cells. Each grid cell

$$V_{\mu,\nu} = (Q, q_0, w_0)_{\mu,\nu} \tag{2}$$

is composed of a finite set of states $Q = \{\square, p_1, \ldots, p_{16}\}$ with the meaning that a grid cell can either be free from any plant ($\square$) or contains one plant out of plant species $p_1$ to $p_{16}$. The initial state $q_0 \in Q$ represents the vegetation in the beginning. The multiset $w_0 \in O^*$ symbolizes the initial facilities found in the grid cell. For example, $w_0 = \{eeeeeeeeees_3s_3s_3s_3h_{2,0}h_{2,0}h_{2,0}h_{2,0}h_{2,0}\} = \{(e, 10), (s_3, 4), (h_{2,0}, 5)\}$ specifies a grid cell whose plant possesses 10 units of energy, releases 4 units of signaling compound $s_3$ and is occupied by 5 individuals of herbivore species $h_2$ whose cluster survives in zero generation. Note that grid cells equipped with energy ($e$), with a bitter substance ($b_j$), or with a toxin ($l_k$) must be in a state that differs from $\square$ that is resided by a plant.

- The set of edges $E \subseteq V_{n,n} \times V_{n,n}$ stands for the underground network (mycorrhiza) that connects plants and hence grid cells for transport of signaling compounds unable to be spread over air. $E$ is allowed to be an empty set in case of no mycorrhiza exists.
- The parameter $t_{\text{fin}} \in \mathbb{N} \cup \{\infty\}$ permits setting of a time limit with respect to the number of time steps running the system.

Let us now construct *rules* subsumed by $\mathcal{R}$ according to the events and actions which may happen. For definition of the rules, we use two auxiliary functions. First, $q_{\mu,\nu} : \mathbb{N} \to Q_{\mu,\nu}$ whereas $q(t)_{\mu,\nu}$ contains the state of grid cell $V_{\mu,\nu}$ at time point $t$. Moreover, $w_{\mu,\nu} : \mathbb{N} \to O^*$ stores the configuration (contents) of grid cell $V_{\mu,\nu}$ at time point $t$. For initialization, we set $q(0)_{\mu,\nu} = q_{0_{\mu,\nu}}$ and $w(0)_{\mu,\nu} = w_{0_{\mu,\nu}}$. Each rule is carried out under given conditions reflecting $q(t)_{\mu,\nu}$ and $w(t)_{\mu,\nu}$ from one or more grid cells. In addition, each rule has a *timer* denoted as a positive natural number on top of the arrow. For instance, the timer $\xrightarrow{5}$ indicates that the rule will be executed after 5 time steps have been elapsed. After each single time step, a check is made whether or not the predefined conditions for execution are still met. If so, the time becomes decreased by one. If not, the rule is dequeued and has no further effect. As soon as the timer has reached 0 and all predefined conditions are still fulfilled, the rule gets conducted.

- Energy increase of a plant due to photosynthesis

  $[\,]_{\mu,\nu} \xrightarrow{1} [\underbrace{e \ldots e}_{a-\text{times}}]_{\mu,\nu}$

  under the condition $q(t)_{\mu,\nu} = p_\kappa$ formulates the increase of energy in the plant of species $p_\kappa$ available in grid cell $V_{\mu,\nu}$ at point in time $t$ due to photosynthesis. One time step after, at $t + 1$, the amount of energy rises by $a$ energy units.
- Elimination of a plant from grid cell due to lack of energy

  Let $a \in \mathbb{N} \setminus \{0\}$ be a threshold for the minimal number of energy units necessary for survival of a plant $p_\kappa$ residing in grid cell $V_{\mu,\nu}$.

  $[\underbrace{e \ldots e}_{b-\text{times}}]_{\mu,\nu} \xrightarrow{1} [\,]_{\mu,\nu} \; \forall b = 1, \ldots, a; \quad q(t+1)_{\mu,\nu} = \square$

  under the condition $(q(t)_{\mu,\nu} = p_\kappa) \;\wedge\; ((w(t)_{\mu,\nu} \setminus \{(e, a)\}) \cap \{(e, \infty)\} = \emptyset)$

eliminates plant $p_\kappa$ from grid cell $V_{\mu,\nu}$ due to lack of energy for survival at time point $t+1$. In addition, remaining energy units have been removed as well at time point $t+1$.

- Removal of remaining bitter substances from a grid cell without plant

  $[\underbrace{b_j \dots b_j}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{1} [\ ]_{\mu,\nu}$ with $a = |w(t)_{\mu,\nu} \cap \{(b_j, \infty)\}|$ and $\forall j = 1, \dots, 15$

  under the condition $q(t)_{\mu,\nu} = \square$ removes all remaining bitter substances from grid cell $V_{\mu,\nu}$ with no residing plant.
- Removal of remaining toxins from a grid cell without plant

  $[\underbrace{l_k \dots l_k}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{1} [\ ]_{\mu,\nu}$ with $a = |w(t)_{\mu,\nu} \cap \{(l_k, \infty)\}|$ and $\forall k = 1, \dots, 15$

  under the condition $q(t)_{\mu,\nu} = \square$ removes all toxins from grid cell $V_{\mu,\nu}$ with no residing plant.
- Reproduction of a plant and placing its offspring

  $[\underbrace{e \dots e}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\underbrace{e \dots e}_{b-\text{times}}]_{\alpha,\beta}[\underbrace{e \dots e}_{c-\text{times}}]_{\gamma,\delta} \dots [\underbrace{e \dots e}_{d-\text{times}}]_{\epsilon,\phi};$

  $q(t+\theta)_{\alpha,\beta} = p_\kappa; q(t+\theta)_{\gamma,\delta} = p_\kappa; \dots; q(t+\theta)_{\epsilon,\phi} = p_\kappa$ under the condition $(q(t)_{\mu,\nu} = p_\kappa) \wedge (q(t)_{\alpha,\beta} = \square) \wedge (q(t)_{\gamma,\delta} = \square) \wedge \dots \wedge (q(t)_{\epsilon,\phi} = \square)$ generates offspring of the plant in $V_{\mu,\nu}$ after $\theta$ time steps. The offspring reside in grid cells $V_{\alpha,\beta}, V_{\gamma,\delta}, \dots, V_{\epsilon,\phi}$ having initial energy of $b$, $c$, …, $d$ units within an environment of their ancestor. Grid cells occupied by offspring need to be free from any plant before.
- Movement of a herbivores cluster searching for a plant

  $[\underbrace{h_{m,\tau} \dots h_{m,\tau}}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\ ]_{\mu,\nu}[\underbrace{h_{m,\tau} \dots h_{m,\tau}}_{a-\text{times}}]_{\alpha,\beta}$

  under the condition $q(t)_{\mu,\nu} = \square$ moves the herbivores cluster $h_{m,\tau}$ with size $a$ (number of individuals) located in grid cell $V_{\mu,\nu}$ to grid cell $V_{\alpha,\beta}$. The process takes $\theta$ time steps.
- Herbivores cluster feeds on a plant
  diminishing plant energy and increasing cluster size

  $[\underbrace{e \dots e}_{a-\text{times}}\ \underbrace{h_{m,\tau} \dots h_{m,\tau}}_{b-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\underbrace{h_{m,\tau} \dots h_{m,\tau}}_{(b+1)-\text{times}}]_{\mu,\nu}$

  under the condition $q(t)_{\mu,\nu} = p_\kappa$ decreases the energy level of the plant located in grid cell $V_{\mu,\nu}$ by $a$ energy units while the number of individuals of herbivores cluster $h_{m,\tau}$ increases by 1. The process takes $\theta$ time steps.
- Plant under attack by at least one herbivores cluster
  produces signaling compounds

  $[\underbrace{e \dots e}_{c-\text{times}}\ \underbrace{h_{1,\tau} \dots h_{1,\tau}}_{a_{1,\tau}-\text{times}} \dots \underbrace{h_{15,\tau} \dots h_{15,\tau}}_{a_{15,\tau}-\text{times}}]_{\mu,\nu} \xrightarrow{\theta}$

  $[\underbrace{s_i \dots s_i}_{b-\text{times}}, \underbrace{h_{1,\tau} \dots h_{1,\tau}}_{a_{1,\tau}-\text{times}} \dots \underbrace{h_{15,\tau} \dots h_{15,\tau}}_{a_{15,\tau}-\text{times}}]_{\mu,\nu}$

with $i \in \{1, \ldots, 15\}$ and $\tau = 0, \ldots, t_{\text{fin}}$
under the condition $q(t)_{\mu,\nu} \neq \square$ formalizes production of signaling compounds from type $s_i$ ($b$ units) after $\theta$ time steps had been elapsed and the plant in grid cell $V_{\mu,\nu}$ is attacked by herbivores. This process consumes $c$ units of energy.

- Signaling compounds spread over air

$$[\underbrace{s_i \ldots s_i}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\ ]_{\mu,\nu}[\underbrace{s_i \ldots s_i}_{a-\text{times}}]_{\alpha,\beta} \text{ with } i \in \{1, \ldots, 15\}$$

under the condition $q(t)_{\mu,\nu} = p_\kappa$ takes $a$ units of signaling compound $s_i$ from plant $p_\kappa$ in grid cell $V_{\mu,\nu}$ and transports them to an arbitrary grid cell $V_{\alpha,\beta}$. It does not matter whether or not a plant is located there. $\theta$ time steps are required to finalize this process.

- Signaling compounds propagate through mycorrhiza

$$[\underbrace{s_i \ldots s_i}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\ ]_{\mu,\nu}[\underbrace{s_i \ldots s_i}_{a-\text{times}}]_{\alpha,\beta} \text{ with } i \in \{1, \ldots, 15\}$$

under the conditions $q(t)_{\mu,\nu} = p_\kappa$ and $q(t)_{\alpha,\beta} = p_\lambda$ and $(V_{\mu,\nu}, V_{\alpha,\beta}) \in E$ transports $a$ units of signaling compound $s_i$ from plant $p_\kappa$ in grid cell $V_{\mu,\nu}$ to plant $p_\lambda$ located in grid cell $V_{\alpha,\beta}$. Both plants need to be directly connected via mycorrhiza. The process takes $\theta$ time steps.

- Degradation of signaling compounds

$$[\underbrace{s_i \ldots s_i}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\ ]_{\mu,\nu}$$

without any condition decreases the amount of signaling compound $s_i$ in grid cell $V_{\mu,\nu}$ by $a$ units. The process takes $\theta$ time steps.

- Production of lethal toxin in the presence of signaling compounds

$$[\underbrace{e \ldots e}_{a-\text{times}}\ \underbrace{s_1 \ldots s_1}_{a_1-\text{times}} \ldots \underbrace{s_{15} \ldots s_{15}}_{a_{15}-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\underbrace{s_1 \ldots s_1}_{a_1-\text{times}} \ldots \underbrace{s_{15} \ldots s_{15}}_{a_{15}-\text{times}}\ \underbrace{l_k \ldots l_k}_{c-\text{times}}]_{\mu,\nu}$$

with $k \in \{1, \ldots, 15\}$
under the condition $q(t)_{\mu,\nu} = p_\kappa$ produces $c$ units of lethal toxin $l_k$ in plant $p_\kappa$ located in grid cell $V_{\mu,\nu}$ in the presence of specified signaling compounds $s_1$ to $s_{15}$. The process takes $\theta$ time steps and consumes $a$ units of energy.

- Production of bitter substance in the presence of signaling compounds

$$[\underbrace{e \ldots e}_{a-\text{times}}\ \underbrace{s_1 \ldots s_1}_{a_1-\text{times}} \ldots \underbrace{s_{15} \ldots s_{15}}_{a_{15}-\text{times}}]_{\mu,\nu} \xrightarrow{\theta} [\underbrace{s_1 \ldots s_1}_{a_1-\text{times}} \ldots \underbrace{s_{15} \ldots s_{15}}_{a_{15}-\text{times}}\ \underbrace{b_j \ldots b_j}_{c-\text{times}}]_{\mu,\nu}$$

with $j \in \{1, \ldots, 15\}$
under the condition $q(t)_{\mu,\nu} = p_\kappa$ produces $c$ units of bitter substance $b_j$ in plant $p_\kappa$ located in grid cell $V_{\mu,\nu}$ in the presence of specified signaling compounds $s_1$ to $s_{15}$. The process takes $\theta$ time steps and consumes $a$ units of energy.

- Diminishing herbivores cluster in the presence of lethal toxin

$$[\underbrace{h_{1,\tau}\dots h_{1,\tau}}_{a_{1,\tau}-\text{times}}\dots\underbrace{h_{15,\tau}\dots h_{15,\tau}}_{a_{15,\tau}-\text{times}}\underbrace{l_1\dots l_1}_{c_1-\text{times}}\dots\underbrace{l_{15}\dots l_{15}}_{c_{15}-\text{times}}]_{\mu,\nu} \xrightarrow{\theta}$$
$$[\underbrace{l_1\dots l_1}_{d_1-\text{times}}\dots\underbrace{l_{15}\dots l_{15}}_{d_{15}-\text{times}}]_{\mu,\nu}$$

with $\tau = 0,\dots,t_{\text{fin}}$ and $d_1 \le c_1,\dots,d_{15} \le c_{15}$
under the condition $q(t)_{\mu,\nu} = p_\kappa$ reduces the number of individuals within herbivores clusters $h_{1,\tau}$ to $h_{15,\tau}$ over all existing cluster generations $\tau$ in the presence of lethal toxins $l_1$ to $l_{15}$. This process takes $\theta$ time steps and happens within grid cell $V_{\mu,\nu}$.

- Repelling herbivores cluster in the presence of bitter substance

$$[\underbrace{h_{1,\tau}\dots h_{1,\tau}}_{a_{1,\tau}-\text{times}}\dots\underbrace{h_{15,\tau}\dots h_{15,\tau}}_{a_{15,\tau}-\text{times}}\underbrace{b_1\dots b_1}_{b_1-\text{times}}\dots\underbrace{b_{15}\dots b_{15}}_{b_{15}-\text{times}}]_{\mu,\nu} \xrightarrow{\theta}$$
$$[\underbrace{b_1\dots b_1}_{c_1-\text{times}}\dots\underbrace{b_{15}\dots b_{15}}_{c_{15}-\text{times}}]_{\mu,\nu}[\underbrace{h_{1,\tau}\dots h_{1,\tau}}_{a_{1,\tau}-\text{times}}\dots\underbrace{h_{15,\tau}\dots h_{15,\tau}}_{a_{15,\tau}-\text{times}}]_{\alpha,\beta}$$

with $\tau = 0,\dots,t_{\text{fin}}$ and $c_1 \le b_1,\dots,c_{15} \le b_{15}$
under the condition $q(t)_{\mu,\nu} = p_\kappa$ diverts herbivores clusters $h_{1,\tau}$ to $h_{15,\tau}$ over all existing cluster generations $\tau$ in the presence of bitter substances $b_1$ to $b_{15}$. Herbivores clusters have been repelled from the plant $p_\kappa$ located in grid cell $V_{\mu,\nu}$. They invade grid cell $V_{\alpha,\beta}$ after $\theta$ time steps. It does not matter whether or not a plant is situated there.

- Removal of remaining lethal toxins after attack by herbivores cluster

$$[\underbrace{l_k\dots l_k}_{a-\text{times}}]_{\mu,\nu} \xrightarrow{1} [\ ]_{\mu,\nu} \text{ with } a = |w(t)_{\mu,\nu} \cap \{(l_k,\infty)\}| \text{ and } \forall k = 1,\dots,15$$

under the conditions $w(t)_{\mu,\nu} \cap \{(h_{1,\tau},\infty),(h_{2,\tau},\infty),\dots,(h_{15,\tau},\infty) \mid \tau = 0,\dots,t_{\text{fin}}\} = \emptyset$ and $q(t)_{\mu,\nu} = p_\kappa$ removes all lethal toxins from grid cell $V_{\mu,\nu}$ settled by plant $p_\kappa$ but with no residing herbivores cluster. We estimate one time step for this process.

- Removal of remaining bitter substances after attack by herbivores cluster

$$[\underbrace{b_j\dots b_j}_{b-\text{times}}]_{\mu,\nu} \xrightarrow{1} [\ ]_{\mu,\nu} \text{ with } a = |w(t)_{\mu,\nu} \cap \{(b_j,\infty)\}| \text{ and } \forall j = 1,\dots,15$$

under the conditions $w(t)_{\mu,\nu} \cap \{(h_{1,\tau},\infty),(h_{2,\tau},\infty),\dots,(h_{15,\tau},\infty) \mid \tau = 0,\dots,t_{\text{fin}}\} = \emptyset$ and $q(t)_{\mu,\nu} = p_\kappa$ removes all bitter substances from grid cell $V_{\mu,\nu}$ settled by plant $p_\kappa$ but with no residing herbivores cluster. We estimate one time step for this process.

- Division of grown herbivores cluster into two clusters of next generation

Let $size\_initial_{m,0} = \sum_{\mu=1}^{n}\sum_{\nu=1}^{n}\left(|w_{0_{\mu,\nu}} \cap \{(h_{m,0},\infty)\}|\right)$ the initial size of herbivores cluster $h_{m,0}$ (zero generation).

$$[\underbrace{h_{m,\tau}\dots h_{m,\tau}}_{2\cdot size\_initial_{m,0}}]_{\mu,\nu} \xrightarrow{1} [\underbrace{h_{m,\tau}\dots h_{m,\tau}}_{size\_initial_{m,0}}\underbrace{h_{m,\tau+x}\dots h_{m,\tau+x}}_{size\_initial_{m,0}}]_{\mu,\nu}$$

whereas $x \in \mathbb{N}$ and $x > 0$ and $\tau + x \le t_{\text{fin}}$ is arbitrarily but uniquely chosen indicating a new generation number.

## 5 Implementation, Graphical Interface, User Manual

Since existing simulation tools turn out to be inexpedient for our purpose due to missing features and due to insufficient parameterization, we prefer implementation of a new software instead of enhancing a tool with much more effort. The *Python* programming language, version 3.12.5, was chosen to implement the simulator software. The simulator's implementation uses as software architecture the *Model–View–Controller* (MVC) pattern. For dependency management, we employed *poetry*[5], ensuring the reproducibility of the working environment. The numerical calculations within the model were carried out using the *NumPy* library [17]. For visualizing the course of a simulation, *Matplotlib* [18] was employed to generate diagrams. To prevent simulation results from being lost once the tool is closed, a file management system was implemented. This includes the class `Export`, responsible for creating files and saving the configured networks, as well as the class `Import`, which reconstructs stored networks by reloading a selected file back into the tool.

The graphical user interface consists of three sections, one of which is a control bar used to define the framework conditions for the grid. This includes specifying the grid size as well as setting the number of different actors and substances.

In addition, various buttons are available to control the simulation of the configured network and to execute specific actions. Figure 2 (upper part) shows the four input fields, each of which expects a natural number as input. The other controls shown in the lower part are various buttons that are used to perform different actions.




Fig. 2: Topbar controls and input fields

Once valid inputs are provided in the control bar, an empty grid with the corresponding number of actors is generated. Below the control bar, two horizontally aligned sections are displayed: on the left-hand side, input elements for parameter configuration of the actors and substances; on the right-hand side, the visualization of the grid itself.

The left-hand side for parameter configuration is subdivided into three tabs, see Figure 3. The first one contains the input fields for the various parameters of the defined plant species followed by a tab that provides the input fields for parameters defined by the user for the herbivore species. The third tab is reserved for entering the parameters of the different substances.

[5] https://github.com/python-poetry/poetry

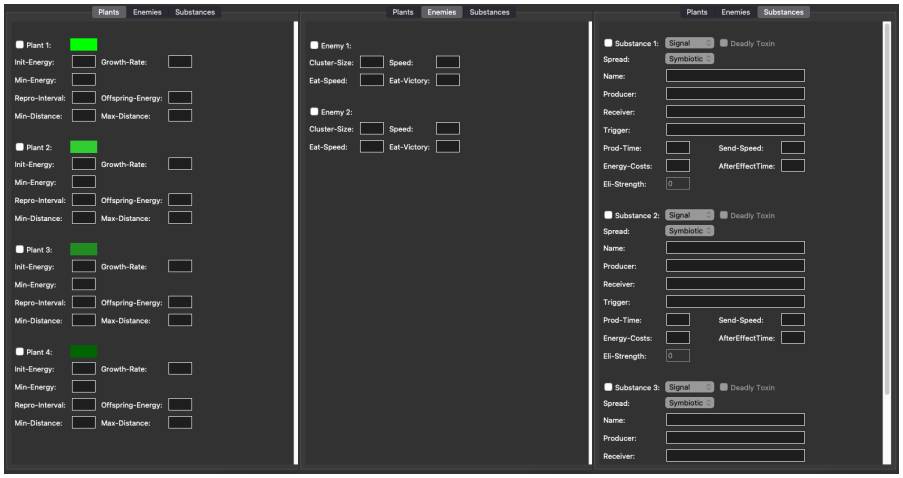



Fig. 3: Tabs for configuration of plants, herbivores, and substances

### 5.1 Initialization of the Tool

When the tool is started, only an empty window with the control bar is displayed. In order to make the parameter tabs and the empty grid visible, the grid size as well as the number of plant species, herbivore species, and substances must be specified via the input fields and confirmed using the *Apply* button.

However, the input fields for a scenario do not accept arbitrary values but are restricted by predefined limits. They only allow integer values within the model boundaries defined in the implementation. The grid size must be in the range of 1 to 80, the number of plant species must be between 1 and 16, and the input values for herbivores and substances may only range from 0 to 15. If a value outside these ranges is entered, the tool issues an error message informing the user that the input was invalid and prompts correction of the parameters by displaying the permissible numerical range.

### 5.2 Functions of the Control Bar

Upon successful input, an empty scenario is created in which the parameters for plants, herbivores, and substances can be individually defined. Actors with completed parameter settings can at this point be placed on the grid.

The *Breakups* button opens a small window in which the different termination conditions for a simulation can be defined. Figure 4 shows, on the one hand, an empty window when no termination conditions are specified (left), and, on the other hand, a window with completed input fields (right). The latter means that a simulation ends if either 1000 time steps have elapsed, all plants of species `p1` have died, all herbivores of species `e1` have died, an upper limit of $10,000$ units of plant energy has been reached, or a maximum of $20,000$ herbivore individuals has been reached.




Fig. 4: Defining the conditions for termination of a simulation run

Next in the control bar is the *Play* button, the function of which is straightforward: it starts the simulation once the user has finished configuring the network and wishes to initiate execution. The right-hand element in the control bar is a color-marked field displaying the current time step of the simulation. Here, a signal color indicates whether the simulation has not yet started, is currently running, or has already ended.

Following this is the *Plot* button. This opens a window in which, for a successfully executed simulation, four different plots representing simulation events can be displayed. To ensure clarity, the plots are organized in tabs between which the user can switch. It is important to note that the plot window is only accessible after a simulation has been executed, since the required information is collected during runtime. Otherwise, the tool blocks access and displays a message instructing the user to first start a simulation before time-series plots can be generated.

The *Import* button opens a window displaying the file manager of the user's device. There, previously exported files can be located and selected for import into the tool. After a file has been selected, all defined parameters are automatically transferred to the corresponding input fields, and the placement of actors on the grid is restored.

The *Export* button also opens a window, allowing a configured network to be saved as a file under a user-defined name. Since the tool's file manager is based on the Python module `pickle`, files are stored in the `.pkl` format.

The final button in the control bar is the *Reset* button. It restores a completed simulation to its initial state, enabling a new simulation run without requiring the network to be reconfigured or re-imported.

### 5.3 Preparing the Grid

In order to place plants on the grid, the corresponding input fields in the tab for the plant species to be placed must first be completed. A plant species is defined by seven different input fields, which are listed in Table 1.

It is important to note that the input field for the reproduction interval may alternatively be left blank instead of entering a 0 if the plant is not intended to reproduce. In this case, the graphical interface interprets the empty field as 0.

| Parameter | Description | Range |
|---|---|---|
| Init-Energy | Initial energy of a plant | $\mathbb{N}$ |
| Energy-Growth | Energy growth | $\mathbb{N}$ |
| Min-Energy | Minimum energy required for survival | $\mathbb{N}$ |
| Repro-Interval | Interval after which offspring are produced | $\mathbb{N} \setminus \{0\}$ |
| Offspring-Energy | Initial energy of the offspring | $\mathbb{N}$ |
| Min-Distance | Minimum dispersal distance for offspring | $\mathbb{N}$ |
| Max-Distance | Maximum dispersal distance for offspring | $\mathbb{N}$ |

Table 1: Parameters of a plant species

Furthermore, for the offspring input fields, it is taken into account that if the maximum dispersal distance is accidentally set to a smaller value than the minimum distance, the maximum distance is automatically adjusted to the value of the minimum distance plus 1. This ensures that the maximum distance is always greater than the minimum distance.

A herbivore species, defined in the herbivore tab, is characterized by four input fields for its parameters, which are listed in Table 2.

| Parameter | Description | Range |
|---|---|---|
| Cluster-Size | Number of individuals in the cluster | $\mathbb{N} \setminus \{0\}$ |
| Speed | Movement speed in time steps | $\mathbb{N}$ |
| Eat-Speed | Eating rate in units of plant energy | $\mathbb{N}$ |
| Eat-Victory | Energy required to generate one offspring | $\mathbb{N} \setminus \{0\}$ |

Table 2: Parameters of a Herbivore Species

For signaling compounds, there are more options and input fields due to the different types. First, the user can decide via a selection menu whether the substance is a signaling compound or a toxin. If it is a signaling compound, the user can then specify whether it propagates through the air or via grid connections.

If it is a toxin, however, the propagation menu is blocked so that no selection can be made. Instead, the checkbox to the right of the substance type becomes active, allowing the user to define whether the toxin has a lethal effect. If the box is not checked, the toxin exerts a repelling effect on herbivores.

Each substance can be assigned an arbitrary name. This input is not subject to any specific requirements and may contain both letters and numbers, entirely according to the user's preference.

The second input field specifies the plant species capable of producing the substance. A plant species is entered as `p` followed by a number between `1` and `16` corresponding to the species. If multiple plant species produce the same substance, several entries can be provided, separated by commas. The same

| Parameter | Description | Signal Substance | Toxin |
|---|---|---|---|
| Prod-Time | Production time in time steps | Yes | Yes |
| Energy-Costs | Energy costs per time step | Yes | Yes |
| Eli-Strength | Mortality rate per time step | No | Yes |
| Send-Speed | Transmission speed in time steps | Yes | No |
| AfterEffectTime | Duration of aftereffect in time steps | Yes | No |

Table 3: Input fields for numerical substance parameters.

principle applies to the next input field, in which the plant species that can receive a signaling compound are defined.

The next input field, which defines the triggers of a substance, offers different options depending on whether the substance is a signaling compound or a toxin.

For a signaling compound, the herbivore species that activate the signal are specified by entering '`e`' followed by a number between `1` and `15`. Since cluster size also plays a role, it is entered after the herbivore species, separated by a comma. This pair of inputs forms a trigger condition that must be met for the signaling compound to be released. If multiple triggers are required, they are separated by semicolons. An accepted input could, for example, be: `e1,1; e2,5; e3,2`

For toxins, a trigger consists of three elements, each separated by a comma. In addition to the trigger elements already used for signaling compounds, the name of the signaling compound must also be specified, and it must be placed at the beginning of the trigger. In other words, toxin triggers are defined by first the signaling compound name, then the herbivore species, and finally the cluster size. As with signaling compounds, multiple triggers are separated by semicolons. An accepted input could, for instance, be: `s1,e1,1; s2,e2,5; s1,e3,2`. Table 3 gives an overview about input fields for numerical substance parameters.

At this stage, all essential components required for placing the agents on the grid have been addressed. In order to position plants or herbivores clusters on the grid, the corresponding checkbox must be activated. Activating the checkbox selects the agent, thereby enabling its placement on the grid.

For substances, the procedure differs, since they are coupled to plants. Substances also provide a checkbox, which must be activated to inform the tool which substances are to be included in the simulation.

Each grid cell consists of two areas, an inner and an outer section, which can be represented visually as in Figure 5.

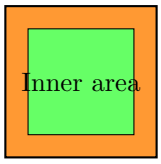



Fig. 5: Inner and outer area of a grid cell

The inner area is responsible for displaying the colors of the selected plants, which fill this section once placed. The outer area, by contrast, indicates the availability of a substance, which is represented in different colors depending on the substance type. The specific colors and their corresponding meanings can be found in Table 4.

| Color | Meaning |
|---|---|
| Yellow | At least one signaling compound is being produced and none is active |
| Dark Orange | At least one signaling compound is active |
| Coral | At least one toxic substance is being produced and none is active |
| Red | At least one toxic substance is active |

Table 4: Meaning of colors for substance activity.

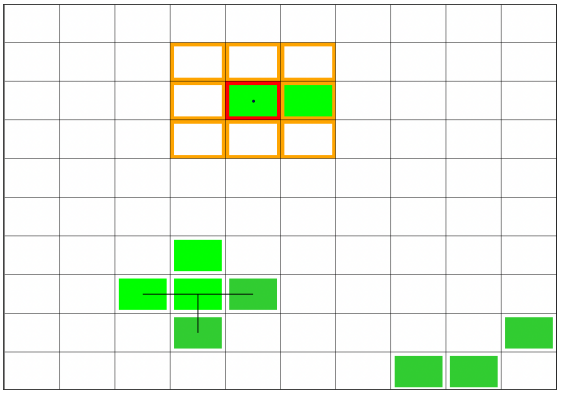


Fig. 6: Example of a predefined grid with plants (dark and light green) and herbivores (black dot). A network of roots connecting neighbored plants is represented by black lines across adjacent cells. Grid cells affected by signaling compounds have been marked with a dark orange outer area while presence of a toxin is framed in red.

For signaling compounds, the radius is visualized by means of an orange highlight applied to the respective fields, with the outer area being filled to indicate to the user where a signaling compound is active. Since multiple signaling compounds can be active on the same grid cell when spreading through the air, the color is rendered progressively darker for each additional substance.

For signaling compounds transmitted via grid connections, the connection must be manually established between plants. This is accomplished by right-clicking on a plant to open a popup window displaying a list of neighboring plants. A connection to a neighboring plant can then be established by activating the corresponding checkbox.

A grid connection is displayed as a black line between neighboring plants. However, once a signaling compound is transmitted through the connection, the color of the line changes to gray.

Herbivores clusters are represented by small markers on the grid cell, which can appear in two different colors. They are shown in dark blue when the herbivores are not lethally poisoned, i.e., when they are in their normal state. If they become lethally poisoned, the marker on the grid is displayed in red.

Within the graphical interface, two mechanisms exist for increasing the size of a cluster. The first method is based on classical growth, in which the number of individuals within a cluster increases through the intake of plant energy. However, this approach results in static simulation behavior. Because herbivores reproduction relies solely on this principle, it is difficult to generate a dynamic interplay between plants and herbivores in which dominance alternates over the course of the simulation.

To increase the dynamism of the simulation, an additional logic has been implemented. This logic checks whether the number of individuals has doubled relative to the initial cluster size. Once this condition is met, the cluster is divided into two separate clusters. In the graphical interface, this is visualized by the appearance of two markers, each representing half of the individuals of the original cluster.

To prevent the number of clusters from growing without limit, the maximum number of clusters is restricted to half of the available grid cells. This constraint enables herbivores to reach a certain size, while still allowing plants a chance to persist and potentially displace the herbivores again. The exact course of the simulation, however, depends primarily on the chosen parameters. Figure 6 illustrates how the grid with plants and herbivores can appear during a simulation.

## 6 Case Study

We conducted a simulation case study to demonstrate the functionality of the tool and to visualize the results using a number of plots. The simulation extends over a longer time horizon, allowing temporal changes to be represented and analyzed in the plots.

### 6.1 Configuration of the Network

To ensure a meaningful dynamic within the simulation, a biotope with a small size of $15 \times 15 = 225$ grid cells is selected. Within this biotope, two different plant species, two herbivore species, and three different substances are present. The agent configuration is based on specific parameters, which are summarized in Table 5. The most significant difference between the two plant species lies in their mode of reproduction: the first species reproduces exclusively locally, whereas the second species disperses its offspring within a larger radius. To balance this difference, the first plant is assigned a shorter reproduction interval than the second. The primary difference between the two herbivore species concerns the energy requirements for reproduction and the energy intake per time step. While the herbivores of the first species require 20 time steps to produce a new individual, the herbivores of the second species demand more energy for reproduction but simultaneously consume more energy per time step. The parameters selected for the substances in the simulation are listed in Table 6.

Plants of the first species pursue a repelling strategy, whereas the plants of the second species employ a lethal strategy. To provide a warning to other

| **Plants** | | | **Herbivores** | | |
|---|---|---|---|---|---|
| **Parameter** | `p1` | `p2` | **Parameter** | `e1` | `e2` |
| Init-Energy | 123 | 90 | Cluster-Size | 30 | 10 |
| Growth-Rate | 3 | 2 | Speed | 1 | 4 |
| Min-Energy | 20 | 10 | Eat-Speed | 1 | 10 |
| Repro-Interval | 30 | 35 | Eat-Victory | 20 | 30 |
| Offspring-Energy | 30 | 50 | | | |
| Min-Distance | 1 | 2 | | | |
| Max-Distance | 2 | 6 | | | |

Table 5: Parameters of the plants and herbivores

| **Parameter** | **Substance 1** | **Substance 2** | **Substance 3** |
|---|---|---|---|
| Type | Signal | Toxin | Toxin |
| Lethal | – | No | Yes |
| Spread | Air | – | – |
| Name | `s1` | `s2` | `s3` |
| Production | `p1,p2` | `p1` | `p2` |
| Receiver | `p1,p2` | – | – |
| Trigger | `e1,10; e2,30` | `s1,e1,20; s1,e2,40` | `s1,e1,40; s1,e2,15` |
| Prod-Time | 3 | 3 | 2 |
| Send-Speed | 10 | – | – |
| Energy-Costs | 1 | 1 | 1 |
| AfterEffectTime | 1 | – | – |
| Eli-Strength | – | – | 6 |

Table 6: Parameters of the substances

plants, a signal substance is dispersed through the air, with its effective radius increasing every 10 time steps. For the initial population of the grid, a diverse mixture of the two plant species and the two herbivore species was selected in order to simulate a realistic biotope, see Figure 7. The plants were arranged in different constellations: adjacent to individuals of the same species and adjacent to individuals of the other plant species. A similar approach was applied to the herbivores, which were evenly distributed across the grid.

### 6.2 Simulation Results

The energy of each plant species was recorded in order to represent it as aggregated time-series curves. To achieve this, all plants of the respective species were iterated over and, for each time step, the currently available energy was summed. The aggregated values are visualized in Plot 8a as a bundled line. The number of plants of each species was determined at every point in time, see Plot 8b. It shows how many plants of a given species are present on the grid at each time step. In addition, the total number of all plants on the grid is also depicted by a black trajectory.

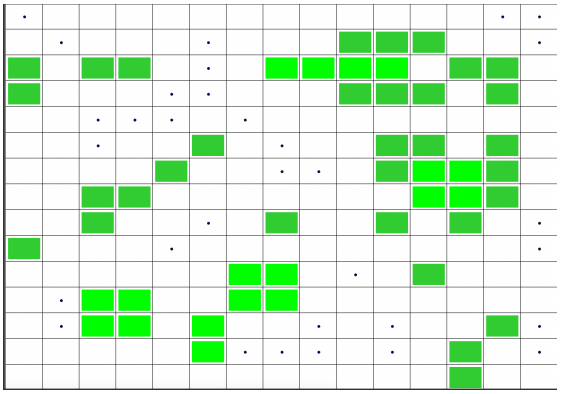
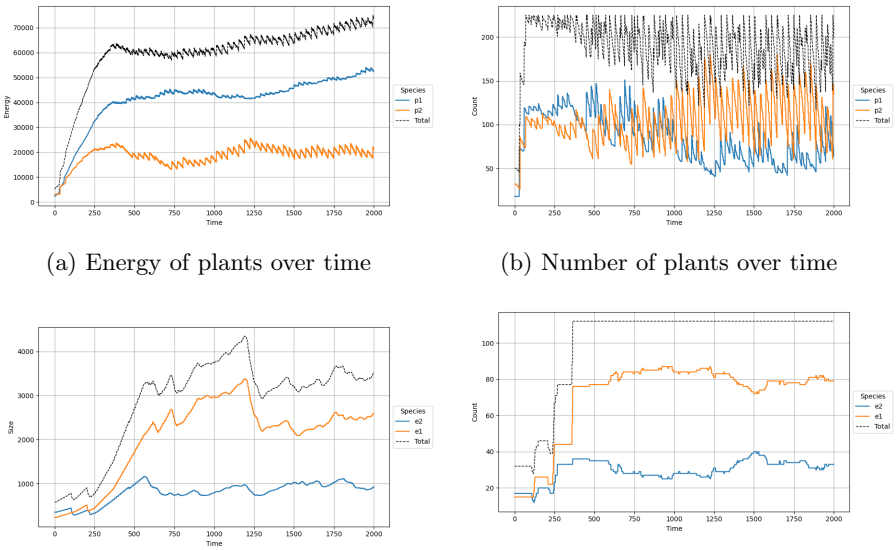


Fig. 7: Initial population of the grid

Plot 8c exhibits the number of individuals of a specific species across all placed herbivores clusters over the course of time while Plot 8d shows the number of clusters of different herbivore species over time. The results clearly demonstrate alterations in the populations of plants and herbivores over time. To obtain meaningful insights, the simulation should run for at least 1000 steps, as all plots exhibit an initial transient phase. Its duration depends on the initial configuration. Longer simulation periods generally lead to more distinct results. Now, we extend the number of simulation time steps to 4000 in order to evaluate the volatility over a longer period of observation.

Plot 9a depicts the energy of the plants. A slightly increasing trend becomes visible indicating that the availability of energy for the plants is subject to temporal fluctuations. The results concerning the number of plants of a specific species are even more evident here. Since the maximum total number of plants is restricted to 225, an inverse division can be observed between species. This means that an increase in the population of one species corresponds to a decrease in the population of the other species. The plots showing the number of individuals of a specific herbivore species likewise reveal an initial adaptation phase, during which the population level adjusts to the given conditions. Subsequently, the number of individuals oscillates around an approximately constant level. These fluctuations depend, among other factors, on which plants the herbivores preferentially consume and how the habitat is populated with plants. In addition, an element of randomness plays a role, as the offspring of plants are placed randomly and their number varies. A closer inspection of Plots 9b and 9c shows that a local minimum is reached in the number of individuals of the herbivore species `e1` around time step 1300, while at the same time the number of plants of species `p2` converges to a local maximum.

The results obtained so far reflect the behavior of an advanced Lotka–Volterra system. In its original form, the underlying model consists of two differential equations and describes the interactions between one predator and a prey [3]. The maximum possible number of herbivores clusters on the grid remains nearly




(a) Energy of plants over time

(b) Number of plants over time

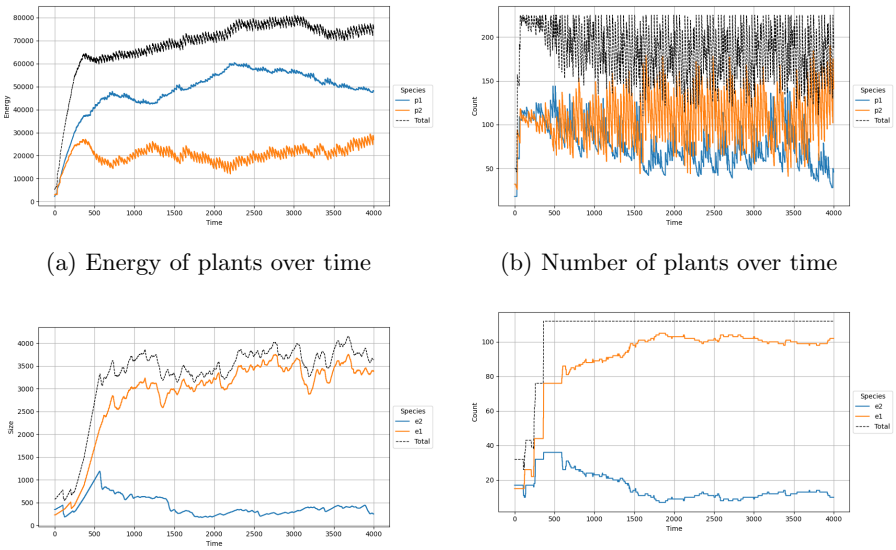
(c) Individuals of herbivores over time

(d) Number of herbivores clusters over time

Fig. 8: Simulation over 2000 time steps

constant after the work-up phase, see Plot 9d. This results from the energy conversion rate throughout the entire system. The process of cluster division along with cluster growth is approximately balanced with the extinction of clusters by toxins. Another noteworthy aspect is the fact that the progression of a simulation highly depends on the chosen parameters. It has been shown that even small modifications to a single parameter can lead to significant changes in the simulation outcomes. To demonstrate this sensitivity, we slightly decrease one single parameter, the trigger threshold for the lethal substance, which is now activated at 35 individuals of herbivore species `e2`, instead of 15 as in the simulation run before.

The effects of this small change become clearly visible. Herbivore type `e1` can no longer sustain itself over the long term and dies out relatively quickly. At the same time, herbivore type `e2` becomes established and exhibits periodic behavior, as can be observed in Plot 10c. Moreover, the total amount of energy available in the plants diminishes in the long term, see Plot 10a. The number of plants also undergoes a slightly decreasing trend (Plot 10b). What stands out is that a variety of system's behavioral patterns might be described by corresponding fine-grained settings of parameter configurations in terms of a multi-dimensional optimization task. During the simulation run, the grid including hosted plants and herbivores clusters undergoes a successive progression over time that is visualized on the fly along with each discrete time step. In this way,




(a) Energy of plants over time

(b) Number of plants over time

(c) Individuals of herbivores over time

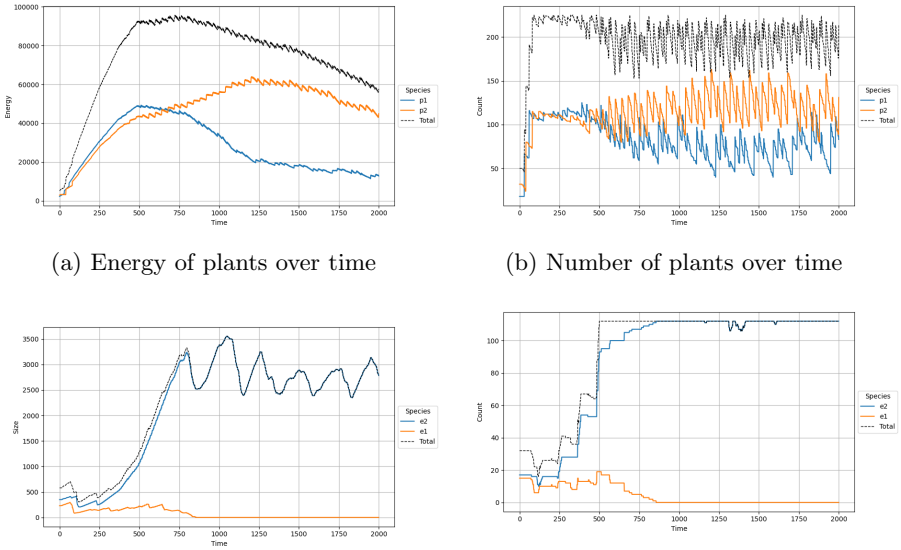
(d) Number of herbivores clusters over time

Fig. 9: Simulation over 4000 time steps

dynamical effects of processes become visible, and correlations between reasons and resulting outcome can be understood in detail.

The configuration of plant-herbivore systems that can persist for a *long time* turns out to be a demanding but productive task. A subsequent analysis of those systems might reveal the essential prerequisites for a setting of plants that is robust and capable of regeneration in an appropriate amount of time. Spatial distributions of plants in a landscape result from upstream evolutionary and behavioral optimization caused by accumulated experiences with numerous herbivores over time. Our simulation software enables an *in-silico* experimental verification of successful and robust spatial arrangements of plants in conjunction with a well-balanced mixture of signaling compounds, bitter substances, and toxins.

In the scenario addressed in the case study, two types of plant species have been taken into account. On the one hand, slowly growing plants able to produce highly effective toxins and on the other hand, rapidly growing plants with limited capacities in production of toxins. Both types of plants interact symbiotically, as slowly growing plants have been closely surrounded by rapidly growing plants. In this combination, herbivores need to invest much more energy and effort to sustainably damage both types of plants. Moreover, fast rehabilitation after a herbivore attack seems feasible. *Insulation spots* spread over the landscape with a specific choice of plant types forming a spot could be the clue to combine




(a) Energy of plants over time

(b) Number of plants over time

(c) Individuals of herbivores over time

(d) Number of herbivores clusters over time

Fig. 10: Simulation over 2000 time steps with merely one slight parameter modification: The lethal toxin is now activated at a higher threshold of 35 individuals from herbivore species `e2` instead of 15.

effective defense with low energy costs. This principle extends protection by simple neighborhood watch.

## 7 Discussion and Conclusions

In terms of natural computation, we adopt most relevant behavioral patterns of plants for protection against herbivores. These patterns include spatial distribution of plants on the ground, molecular signaling mechanism over air or through a symbiotic network of roots when attacks occur, and the production of highly effective toxins. They act in a way either to kill herbivores or to repel them. Our simulation software *PlantProtectionSim* enables a detailed and flexible grid configuration together with initial assignments with plants and herbivores of different types. Furthermore, substances might be specified to form signaling molecules, bitter compounds, or toxins. Due to the high degree of configurability, the tool can capture a large variety of scenarios to be considered as topological optimization processes. All underlying elementary events, like reproduction of a single plant or spreading of signaling molecules for the duration of a time step, form elementary operations on a moderate level of abstraction. The energy of plants and the number of individuals in herbivores clusters serve as

fitness measures to indicate survival and extinction. The visual representation of each simulation run along with the generation of diagrams and charts for analysis illustrates the dynamics of the system under study.

Our simulation software intends to conduct numerous simulation case studies with varying configuration scenarios taken from real life or from desired outcome to find corresponding initial data to produce and reproduce a specific result. The behavioral patterns with all events and actors can be seen as a form of computation when progressing in time and space. Technical applications for construction of secure wireless sensor networks have been envisioned.

From a more general point of view, our approach based on a fine-grained grid with progression of successive discrete events provides a flexible simulation environment for a plethora of case studies beyond those covered by differential equations from the conventional Lotka-Volterra form. We strive for identification of critical parameters and corresponding values crucial for the behavioral pattern as a whole. Especially tipping points and their approximation are of special interest in order to understand what makes the difference between survival and extinction over the long term.

Further work will take into consideration the integration of wind and its direction since distribution of molecules in the air is influenced by airstream, and some herbivores did adapt by feeding contrarily. Moreover, we plan an extension towards an artificial evolution able to achieve optimal grid topologies and settings from scratch with high robustness against specific forms of attacks.

## References

1. A. Alhazov, R. Freund, A. Leporati et al. (Tissue) P systems with unit rules and energy assigned to membranes. *Fundamenta Informaticae* **74(4)**:391-408, 2006
2. Z. Aljbory, M.S. Chen. Indirect plant defense against insect herbivores: a review. *Insect Sci* **25(1)**:2–23, 2018
3. M.C. Anisiu. Lotka, Volterra and their model. *Didactica Mathematica* **32(01)**, 2014
4. F. Baluska (ed.) *Signaling and Communication in Plants.* Springer, 2022
5. S.C. Bhatla, M.A. Lal, *Plant Physiology, Development and Metabolism.* Springer, 2018
6. P. Bhattacharyya, D. Jha, Mycorrhizal Symbiosis in the formation of Antioxidant compounds, *Cabi*, pp. 252–281, 2015
7. J.D. Blande, R. Glinwood (eds.) *Deciphering Chemical Language of Plant Communication.* Springer, 2022
8. T. Bruce. Interplay between insects and plants: dynamic and complex interactions that have coevolved over millions of years but act in milliseconds. *Journal of Experimental Botany* **66(2)**:455–465, 2015
9. M. Burow, B.A. Halkier. How does a plant orchestrate defense in time and space? Using glucosinolates in Arabidopsis as case study. *Current Opinion in Plant Biology* **38**:142–147, 2017
10. M.M. Carreiro, Y.C., Song, J. Wu (eds.) *Ecology, Planning, and Management of Urban Forests.* Springer, 2008

11. S. Chakraborty, J. Gershenzon, S. Schuster. Selection pressure by specialist and generalist insect herbivores leads to optimal constitutive plant defense. A mathematical model. *Ecology and Evolution* **13**, 2023
12. M.S. Chen. Inducible direct plant defense against insect herbivores: a review. *Insect science* **15(2)**:101–114, 2008
13. E.E. Farmer. Surface-to-air signals. *Nature* **411(6839)**:854–856, 2001
14. B. Förster, P. Langendörfer, T. Hinze. Novel Approach to a Plant Inspired Distributed Security Scheme for Wireless Sensor Networks. In: *12th Mediterranean Conference on Embedded Computing (MECO2023)*, pp 1–6, 2023
15. B. Förster, P. Langendörfer, T. Hinze. Determining Distributions of Security Means for WSNs Based on the Model of a Neighborhood Watch. *IEEE Access* pp. 74343–74366, 2024
16. J.A. Gatehouse. Plant resistance towards insect herbivores: a dynamic interaction. *New Phytologist* **156(2)**:145–169, 2002
17. C. Harris, K. Millman, S. van der Wal et al. Array programming with NumPy. *Nature* **585(7825)**:357–362, 2020
18. J.D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* **9(3)**:90–95, 2007
19. S. Lev-Yadun. Aposematic Coloration in Thorny, Spiny and Prickly Plants. *Springer International Publishing*, Cham, pp 119–134, 2016
20. C. Martin-Vide, G. Paun, J.P.A. Rodriguez-Paton. Tissue P Systems. *Theoretical Computer Science* **296**:295-326, 2003
21. C. Pieterse, C. Zamioudis, R. Berendsen et al. Induced Systemic Resistance by Beneficial Microbes. *Annual review of phytopathology* **52**:347–375, 2014
22. R.H. Poppenga. Poisonous plants. In: *Molecular, Clinical and Environmental Toxicology*, Springer, pp. 123–175, 2010
23. B. Song, K. Li, D. Orellana-Martin, X. Zeng, M.J. Perez-Jimenez. Tissue P systems with states in cells. *IEEE Transactions on Computers* **72(9)**:2561-2570, 2023
24. S. Sopory. *Sensory Biology of Plants.* Springer, 2019
25. T. Sun, Y. Zhang. Short-and long-distance signaling in plant defense. *The Plant Journal* **105(2)**:505–517, 2021
26. C.M.F. Vos, K. Kazan (eds.) *Belowground Defence Strategies in Plants.* Springer, 2016
27. I.A. Vos, C. Pieterse, S.C.M. van Wees. Costs and benefits of hormoneregulated plant defences. *Plant Pathology* **62(S1)**:43–55, 2013
28. G. Zhang, M. Perez-Jimenez, A. Riscos-Nunez et al. *Membrane Computing Models: Implementations.* Springer, 2021

# Control Languages Associated with Cell-like Spiking Neural P Systems

Ajeesh Ramanujan[1], Atulya Nagar[2], Rodica Ceterchi[3], K.G. Subramanian[2,*], and Marian Gheorghe[3,4]

[1] Department of Computer Science and Engineering,
College of Engineering Trivandrum, India,
Email: ajeesh@cet.ac.in
[2] School of Mathematics, Computer Science and Engineering,
Liverpool Hope University, Hope Park, Liverpool L16 9JD, UK
([*] Honorary Visiting Professor; Email: kgsmani1948@gmail.com)
[3] Faculty of Mathematics and Computer Science, University of Bucharest,
14 Academiei Street, 010014, Bucharest, Romania,
Email: r.ceterchi@yahoo.com
[4] School of Computing and Engineering, University of Bradford,
West Yorkshire, Bradford BD7 1DP, UK,
Email: m.gheorghe@bradford.ac.uk

**Abstract.** In this paper some control languages associated with cell-like spiking neural P systems are investigated, showing, for two classes of such languages, their relationships with regular and context-free languages. Applications to chain-code picture languages and control-flow modeling are presented.

**Keywords:** Cell-like Spiking Neural P Systems; Control Languages; Chain-Code Picture Languages; Control-Flow Modeling.

## 1 Introduction

The field of Membrane Computing, introduced by Gheorghe Păun [17], has emerged as a prominent area of natural computing that abstracts computational paradigms from the structure and functioning of living cells [18, 20]. Among its various models, spiking neural P systems (SN P systems) have gained considerable attention due to their close resemblance to biological neurons and their ability to process information through discrete spike-based communication [13].

SN P systems are distributed and parallel computing models in which neurons communicate by sending spikes according to predefined firing and forgetting rules. These systems are inherently non-deterministic and maximally parallel, which contributes to their high computational power. However, this very parallelism and non-determinism often make it difficult to impose structured or sequential control over computations, limiting their applicability in scenarios requiring predictable and programmable behavior.

To address this limitation, the concept of control mechanisms has been introduced into membrane computing. One such powerful approach is the use of control languages, wherein each rule of a system is associated with a label, and the sequence of applied rule labels during a computation forms a word over a finite alphabet. The set of all such words constitutes the control language of the system. By restricting computations to those whose corresponding sequences belong to a specified formal language, one can effectively regulate the behavior of the system.

The integration of control languages into SN P systems establishes a strong connection between membrane computing and formal language theory. In particular, it enables the characterization of computational processes in terms of well-known language classes such as regular, context-free, context-sensitive, and recursively enumerable languages. This perspective not only enhances the theoretical understanding of SN P systems but also provides a framework for comparing their expressive power with classical computational models.

Several studies, including those in [14, 21], have investigated the role of control languages in determining the computational capabilities of SN P systems. These works demonstrate that the nature of control – such as restrictions on rule application or the inclusion of silent ($\lambda$-labeled) steps – significantly influences the class of languages that can be obtained. In particular, it has been shown that while restricted control mechanisms yield language classes between regular and context-sensitive, the inclusion of $\lambda$-steps leads to computational completeness equivalent to Turing machines.

Despite these advances, several open problems remain. These include identifying minimal control conditions for universality, analyzing the trade-offs between parallelism and controllability, and exploring efficient implementations of controlled systems. Furthermore, there is growing interest in extending these models to incorporate learning-based or adaptive control mechanisms, thereby bridging the gap between theoretical models and practical applications.

In this paper, we further investigate the role of control languages in cell-like SN P systems. We investigate control languages associated with cell-like SN P systems, showing, for two classes of such languages, their relationships with regular and context-free languages. Applications to chain-code picture languages and control-flow modeling are presented.

## 2 Control Languages Associated with Cell-like SN P Systems: Definitions and an Example

### 2.1 Preliminaries

We assume that the reader is familiar with the theory of formal languages and automata, computability, and membrane computing [22, 20].

Throughout the paper, we use the following notions and notations. Let $V$ be an alphabet and $V^*$ be the set of all words over $V$ (including the empty word, $\lambda$). The length of a word $w \in V^*$ is denoted by $|w|$ and the number of occurrences of the symbol $a \in V$ in $w$ by $|w|_a$.

The families of regular, context-free, context-sensitive and recursively enumerable languages are denoted by $REG$, $CF$, $CS$ and $RE$, respectively.

### 2.2 Basic definitions and an example

We recall the notion of a cell-like SN P system [26], but we do not include the output compartment as it has no role in our subsequent sections. Throughout the entire paper we will refer to a component of a membrane system as compartment or membrane.

**Definition 1.** *A cell-like SN P system (in short, cSN P system), of degree* $m \geq 1$*, is a construct* $\Pi = (O, \mu, n_1, \ldots, n_m, R_1, ..., R_m)$*, where* $O = \{a\}$ *(object* $a$ *is called spike),* $\mu$ *is a hierarchical membrane structure with* $m$ *membranes,* $n_i, 1 \leq i \leq m$*, is the number of spikes present in compartment* $i$ *of* $\mu$ *at the beginning of the computation,* $R_i, 1 \leq i \leq m$*, is the finite set of rules from compartment* $i$*. Besides forgetting rules of the form* $a^s \to \lambda, s \geq 1$*, the sets* $R_i$ *contain spiking rules of the (extended) form* $E/a^c \to u$*, where* $E$ *is a regular expression over* $O, c \geq 1$*, and* $u$ *is a sequence of ordered pairs of the form* $(a^p, tar)$*, where* $p \geq 1$ *and* $tar$ *is a target indication specifying the destination of the* $p$ *associated spikes. This target can be here, out, in,* $in_j$*, where* $j$ *is the label of a membrane, with the usual meaning in cell-like P systems, or* $in_{all}$*, with the meaning that the* $p$ *spikes will be sent, replicated, to all immediately inner membranes (each of them receiving* $p$ *spikes). Of course, in the case of non-extended rules, when only one spike is produced by a rule, only one ordered pair of the form* $(a, tar)$ *will be used. Subsequently, it will be assumed that* $c \geq p \geq 1$*.*

As mentioned before there is no output compartment in this definition.

The computations in a cSN P system are defined as in usual SN P systems: (at most) one rule in each compartment is applied, but the compartments work in parallel, synchronously.

We introduce the notion of labeled cSN (LcSN) P system.

**Definition 2.** *A LcSN P system of degree* $m \geq 1$*, is a construct* $cN = (\Pi, \Sigma, \mathcal{C}, lab)$*, where the cSN P system* $\Pi = (O, \mu, n_1, \ldots, n_m, R_1, \ldots, R_m)$ *is as in the Definition 1,* $\Sigma$ *is a finite input alphabet,* $\mathcal{C}$ *is a finite set of final configurations. Denoting by* $R$ *the set of all the rules associated with the* $m$ *membranes, the labeling function* $lab$ *is given by* $lab : R \to \Sigma \cup \{\lambda\}$*.*

The initial configuration of the system is $(n_1, n_2, \ldots, n_m)$. A transition from one configuration $c_1$ to another configuration $c_2$ of the system, denoted by $c_1 \Rightarrow^b c_2, b \in \Sigma$, is obtained by using only rules, in membranes, with the same label $b$ and rules labeled with $\lambda$. We call such a transition as label restricted and with such a label restricted transition we associate the symbol $b$ if at least one rule with label $b$ is used; if all the rules used have the label $\lambda$, then we associate $\lambda$ to this transition. We refer to the case when all the rules used have $\lambda$ label, as a $\lambda$ step. In a label restricted transition, the input symbol $b$ is read or "consumed" if at least one rule with label $b$ is used. If all the rules used have the label $\lambda$,

then the input symbol is not consumed. Thus, a computation in the system starts from the initial configuration and proceeds through transitions and the associated input word is referred to as a control word.

An input (control) word of symbols over $\Sigma$ is said to be associated with a cSN P system if all its symbols are consumed and the system $\Pi$ halts in a final configuration in the set $\mathcal{C}$. The set of all such strings in a LcSN P system $cN$ is denoted by $L_\lambda(cN)$ with the subscript $\lambda$ indicating the fact that $\lambda$ steps are permitted. When only steps where at least one rule with a non-empty label is used, the accepted language is denoted by $L(cN)$. The family of languages $L(cN)$ associated with LcSN P systems $cN$ with at most $m$ membranes and using forgetting rules and a set $Tar$ of target indicators ($Tar \subseteq \{here, out, in, in_j, in_{all}\}$) is denoted by $LLcSNP_m(for, Tar)$. When forgetting rules are not applied then $for$ is omitted. If the number of membranes is unbounded, then the subscript $m$ is replaced with the $*$ symbol. In the unrestricted case, the corresponding language family is denoted by $L_\lambda LcSNP_m(for, Tar)$.

We illustrate with an example.

*Example 1.* We construct a LcSN P system $cN_1 = (\Pi_1, \Sigma, \mathcal{C}, lab)$ computing the regular language $\{(bc)^n \mid n \geq 1\}$.
$\Pi_1 = (O, \mu, n_1, n_2, R_1, R_2)$, $\Sigma = \{b, c\}$, $\mathcal{C} = \{(0, 0)\}$, $O = \{a\}$, $\mu = [_1[_2]_2]_1$, $n_1 = 0, n_2 = 1$. The sets $R_1, R_2$ contain the labeled rules of the membranes 1 and 2; the rules with their labels are shown in Fig. 1.

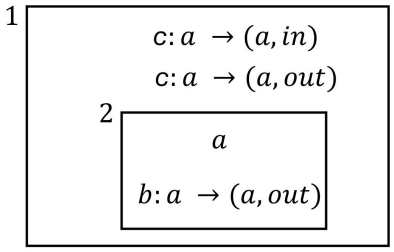



**Fig. 1.** cSN P system $cN_1$ computing $\{(bc)^n \mid n \geq 1\}$

A computation leading to a halting final configuration using a control word of the form $(bc)^n$ can be seen as follows: The initial configuration is $(0, 1)$. The firing rule with label $b$ in membrane 2 is only applicable as the first symbol of the control word is $b$. The application of the rule $b : a \to (a, out)$ sends one spike to the membrane 1 and the resulting configuration is $(1, 0)$. If the next input symbol is $b$, the computation cannot proceed. But the next input symbol of the control word is $c$ and so one of the firing rules in membrane 1 can be applied. If the rule $c : a \to (a, in)$ is applied, one spike is sent to the membrane 2 so that the configuration becomes $(0, 1)$ again. The process can repeat with

the configuration alternating between $(1,0)$ and $(0,1)$. The computation halts on applying the rule $c : a \to (a, out)$ when reading the last symbol $c$ of the control word and the configuration reached is the final configuration $(0,0)$. We have $L(cN_1) \in LLcSNP_2(\{out, in\})$, but one can replace $in$ by any of $in_j, in_{all}$.

## 3 Main Results

The results below provide relationships between classes $LLcSNP_m(Tar)$, for two ways of choosing $Tar$, and regular and context-free languages.

**Theorem 1.** *(i)* $REG \subset LLcSNP_4(\{out, in_j\})$;
*(ii)* $REG \subset LLcSNP_2(\{here, out, in\})$.

*Proof.* Let $L$ be a regular language generated by a regular grammar $G = (N, T, S, P)$ with $N = \{A_1, A_2, \cdots, A_n\}$, $S = A_1$ and the rules in $P$ of the form $A_i \to bA_j, A_i \to b$, $A_i, A_j \in N, b \in T$.

(i) We construct a LcSN P system $cN$ with four membranes and using rules with target indicators $out$ and $in_j$ computing the language $L$. The proof is motivated by the construction in the proof of Theorem 6.1 in [19].

$cN = (\Pi, \Sigma, \mathcal{C}, lab)$, where $\Pi = (O, \mu, n_1, \ldots, n_4, R_1, \ldots, R_4)$, $\Sigma = T$, $\mathcal{C} = \{(0, 2n, 2n, 0), (2n, 0, 0, 2n)\}$, $O = \{a\}$, $\mu = [_1[_2[_4]_4]_2[_3]_3]_1$, $n_1 = n_4 = 2n+1, n_2 = n_3 = 0$. Let $J = \{1, 2, \ldots, n\}$. The rules of $R_i, 1 \leq i \leq 4$, are shown below with their labels.
$R_1 = \{b : a^{2n+i} \to (a^j, in_2)(a^{2n+1}, in_3) \mid A_i \to bA_j \in P, i, j \in J\} \cup \{b : a^{2n+i} \to (a^{2n}, in_3) \mid A_i \to b \in P, i \in J\}$,
$R_2 = \{b : a^{2n+i} \to (a^j, out)(a^{2n+1}, in_4) \mid A_i \to bA_j \in P, i, j \in J\} \cup \{b : a^{2n+i} \to (a^{2n}, in_4) \mid A_i \to b \in P, i \in J\}$.
$R_3 = R_4 = \{\lambda : a^{2n+1} \to (a^{2n}, out)\}$.

Any computation in $cN$ starts from the initial configuration $(n_1, n_2, n_3, n_4) = (2n+1, 0, 0, 2n+1)$, applying from membranes 1 and 4 any of the rules with $a^{2n+1}$ on the left hand (corresponding to rules from $G$ with $A_1 = S$ on their left hand). Any configuration is either $c_1 = (2n+i, 0, 0, 2n+1)$ or $c_1' = (0, 2n+i, 2n+1, 0)$, $i \in J$. In $c_1$ rules with $a^{2n+i}$ on the left side and labeled $b$ in membrane 1 and $\lambda : a^{2n+1} \to (a^{2n}, out)$ in membrane 4 are applicable. The rule in membrane 1 can be either of the form $b : a^{2n+i} \to (a^j, in_2)(a^{2n+1}, in_3)$, $i, j \in J$, or $b : a^{2n+i} \to (a^{2n}, in_3)$, $i \in J$. When the first rule from membrane 1 and the rule from membrane 4 are applied one gets the transition $c_1 \Rightarrow^b c_2 = (0, 2n+j, 2n+1, 0)$, $j \in J$. Obviously, $c_1 \Rightarrow^b c_2$, iff the rule $A_i \to bA_j$ is applied in $G$. When the second rule in membrane 1 and the rule in membrane 4 are applied then we have the transition $c_1 \Rightarrow^b c_2 = (0, 2n, 2n, 0)$, which is a final configuration and the computation stops. This corresponds to a derivation step in $G$ where the rule $A_i \to b$ is applied. Similarly one can show what is the transition from $c_1'$, where the rules applicable are from membranes 2 and 3. Hence, $L(G) = L(cN)$ and consequently $REG \subseteq LLcSNP_4(\{out, in_j\})$.

(ii) To show the second inclusion, one considers a LcSN P system $cN'$ with two membranes and using rules with target indicators *here*, *out* and *in*.

$cN' = (\Pi', \Sigma, \mathcal{C}', lab')$, where $\Pi' = (O, \mu', n_1', n_2', R_1', R_2')$, $\Sigma = T$, $\mathcal{C}' = \{(2n, 2n)\}$, $O = \{a\}$, $\mu' = [_1[_2]_2]_1$, $n_1' = n_2' = 2n + 1$. Let $J = \{1, 2, \ldots, n\}$. The rules of $R_1', R_2'$ are shown below with their labels.
$R_1' = \{b : a^{2n+i} \to (a^j, here)(a^{2n+1}, in) \mid A_i \to bA_j \in P, i, j \in J\} \cup \{b : a^{2n+i} \to (a^{2n}, in) \mid A_i \to b \in P, i \in J\}$,
$R_2' = \{\lambda : a^{2n+1} \to (a^{2n}, out)\}$.

Any computation in $cN'$ starts from $(n_1', n_2') = (2n + 1, 2n + 1)$. As in the previous case, rules from membranes 1 and 2 with the left side $a^{2n+1}$ are applied. Any configuration is of the form $c_1 = (2n + i, 2n + 1)$. In membrane 1 one can apply rules with $a^{2n+i}$ on the left side and label $b$. They have the form either $b : a^{2n+i} \to (a^j, here)(a^{2n+1}, in)$, $i, j \in J$, or $b : a^{2n+i} \to (a^{2n}, in)$, $i \in J$. In membrane 2 the rule that is applied is $\lambda : a^{2n+1} \to (a^{2n}, out)$. If the first rule in membrane 1 and the rule in membrane 2 are applied one gets $c_1 \Rightarrow^b c_2 = (2n + j, 2n + 1)$. Otherwise, when the second rule in membrane 1 and the one in membrane 2 are applied, then we have $c_1 \Rightarrow^b c_2 = (2n, 2n)$, which is a final configuration. Hence, $L(G) = L(cN')$ and consequently $REG \subseteq LLcSNP_2(\{here, out, in\})$.

A computation in each of the LcSN P systems described above is illustrated with an example given after the proof.

The proper inclusion in both cases is proved by showing that the non-regular (context-free) language $L_1 = \{b^n c^n \mid n \geq 2\}$ over the alphabet $\{b, c\}$ is the language of control words associated with all label restricted halting computations of the LcSN P systems presented in this Theorem.

(i) Let us consider the LcSN P system with 3 membranes and using target indicators $out$ and $in_j$, $cN_2 = (\Pi_2, \Sigma, \mathcal{C}, lab)$, where $\Pi_2 = (O, \mu, n_1, n_2, n_3, R_1, R_2, R_3)$, $\Sigma = \{b, c\}$, $\mathcal{C} = \{(0, 0, 4)\}$, $O = \{a\}$, $\mu = [_1[_2[_3]_3]_2]_1$, $n_1 = 0, n_2 = n_3 = 3$ and
$R_1 = \{c : a(a^2)^+/a^2 \to (a, out), c : a \to (a, out)\}$,
$R_2 = \{b : a^3 \to (a^2, out)(a^3, in_3), b : a \to (a, out)\}$,
$R_3 = \{\lambda : a^3 \to (a^3, out), \lambda : a^3/a^2 \to (a, out)\}$.

Any computation in $cN_2$ starts from the configuration $(n_1, n_2, n_3) = (0, 3, 3)$; applying $b : a^3 \to (a^2, out)(a^3, in_3) \in R_2$ and $\lambda : a^3 \to (a^3, out) \in R_3$ in parallel $n \geq 0$ times, one gets the configuration $(2n, 3, 3)$ and then applying $b : a^3 \to (a^2, out)(a^3, in_3) \in R_2$ and $\lambda : a^3/a^2 \to (a, out) \in R_3$ in parallel, $(2n + 2, 1, 4)$ is obtained. Finally, $b : a \to (a, out) \in R_2$ leads to $(2n + 3, 0, 4)$, $n \geq 0$. All the transitions have label $b$. Hence, in $L(cN_2)$ the subword $b^n$, $n \geq 2$, is obtained. From this configuration one can continue by applying $c : a(a^2)^+/a^2 \to (a, out)$ $n \geq 1$ times and getting $(1, 0, 4)$. Now, the rule $c : a \to (a, out)$ is applied and the final configuration $(0, 0, 4)$ is obtained. All the transitions applied starting with the configuration $(2n+3, 0, 4)$ have label $c$. Hence, $b^n c^n$, $n \geq 2$, is in $L(cN_2)$ and consequently $L(cN_2) = L_1$.

For the case (ii), the LcSN P system with two membranes and using $here, out$ and $in$, $cN_2' = (\Pi_2', \Sigma, \mathcal{C}', lab')$, where $\Pi_2' = (O, \mu', n_1', n_2', R_1', R_2')$, $\Sigma = \{b, c\}$, $\mathcal{C}' = \{(0, 0)\}$, $O = \{a\}$, $\mu' = [_1[_2]_2]_1$, $n_1 = 0, n_2 = 3$ and

$R'_1 = \{c : a(a^2)^+/a^2 \to (a, out), c : a \to (a, out)\}$,
$R'_2 = \{b : a^3 \to (a^3, here)(a^2, out), b : a^3 \to (a, here)(a^2, out), b : a \to (a, out)\}$.

The LcSN P systems $cN_2$ and $cN'_2$ are very similar. The way the rules from $cN_2$, $b : a^3 \to (a^2, out)(a^3, in_3) \in R_2$ and $\lambda : a^3 \to (a^3, out) \in R_3$ that work in parallel in membranes 2 and 3, respectively, is replaced in $cN'_2$ by $b : a^3 \to (a^3, here)(a^2, out) \in R'_2$. Similarly, the way the rules $b : a^3 \to (a^2, out)(a^3, in_3) \in R_2$ and $\lambda : a^3/a^2 \to (a, out) \in R_3$ work in parallel in $cN_2$ is obtained in $cN'_2$ by $b : a^3 \to (a, here)(a^2, out)$. Obviously, $L(cN'_2) = L_1$. □

*Remark 1.* From the proof of Theorem 1 it follows that $cN$ has 4 membranes and $2R+2$ rules in total and $cN'$ contains 2 membranes and $R+1$ rules in total, where $R$ is the total number of rules of the regular grammar $G$.

*Example 2.* Now, we illustrate the constructions of the two LcSN P systems described in the proof of Theorem 1 for the language $L_2$ generated by the regular grammar $G = (N, T, P, S)$, where $N = \{A_i \mid 1 \leq i \leq n\}$, $n = 3$, $T = \{b, c\}$, the rules of $P$ are $A_1 \to bA_2$, $A_2 \to bA_2$, $A_2 \to cA_3$, $A_3 \to cA_1$, $A_1 \to b$, and $S = A_1$. The string $w = bbccb$ is obtained from the derivation:
$S = A_1 \Rightarrow bA_2 \Rightarrow bbA_2 \Rightarrow bbcA_3 \Rightarrow bbccA_1 \Rightarrow bbccb = w$.

(i) We build the LcSN P system as in the proof of Theorem 1, case (i),
$cN_3 = (\Pi_3, \Sigma, \mathcal{C}, lab)$, where $\Pi_3 = (O, \mu, n_1, n_2, n_3, n_4, R_1, R_2, R_3, R_4)$, $\Sigma = \{b, c\}$, $\mathcal{C} = \{(0, 6, 6, 0), (6, 0, 0, 6)\}$, $O = \{a\}$, $\mu = [_1[_2[_4]_4]_2[_3]_3]_1$, $n_1 = 7, n_2 = 0, n_3 = 0, n_4 = 7$.
The set $R_1$ contains
$b : a^7 \to (a^2, in_2)(a^7, in_3), A_1 \to bA_2$; $b : a^8 \to (a^2, in_2)(a^7, in_3), A_2 \to bA_2$;
$c : a^8 \to (a^3, in_2)(a^7, in_3), A_2 \to cA_3$; $c : a^9 \to (a, in_2)(a^7, in_3), A_3 \to cA_1$;
$b : a^7 \to (a^6, in_3), A_1 \to b$.
$R_2$ contains
$b : a^7 \to (a^2, out)(a^7, in_4), A_1 \to bA_2$; $b : a^8 \to (a^2, out)(a^7, in_4), A_2 \to bA_2$;
$c : a^8 \to (a^3, out)(a^7, in_4), A_2 \to cA_3$; $c : a^9 \to (a, out)(a^7, in_4), A_3 \to cA_1$;
$b : a^7 \to (a^6, in_4), A_1 \to b$.
$R_3$ and $R_4$ contain $\lambda : a^7 \to (a^6, out)$.

In $cN_3$ one starts from $(7, 0, 0, 7)$ and go through the sequence of transitions below such that the derivation in $G$ is simulated leading to $w$ :
$(7, 0, 0, 7) \Rightarrow^b (0, 8, 7, 0) \Rightarrow^b (8, 0, 0, 7) \Rightarrow^c (0, 9, 7, 0) \Rightarrow^c (7, 0, 0, 7) \Rightarrow^b (0, 6, 6, 0)$.

One can easily observe how the rules of $cN_3$ corresponding to those of $G$ are used in the above sequence of transitions.

(ii) The following LcSN P system is built as in the proof of Theorem 1, case (ii),
$cN'_3 = (\Pi'_3, \Sigma, \mathcal{C}', lab')$ where $\Pi'_3 = (O, \mu', n'_1, n'_2, R'_1, R'_2)$, $\Sigma = \{b, c\}$, $\mathcal{C}' = \{(6, 6)\}$, $O = \{a\}$, $\mu' = [_1[_2]_2]_1$, $n'_1 = 7, n'_2 = 7$.
The set $R'_1$ contains
$b : a^7 \to (a^2, here)(a^7, in), A_1 \to bA_2$; $b : a^8 \to (a^2, here)(a^7, in), A_2 \to bA_2$;
$c : a^8 \to (a^3, here)(a^7, in), A_2 \to cA_3$; $c : a^9 \to (a, here)(a^7, in), A_3 \to cA_1$;
$b : a^7 \to (a^6, in), A_1 \to b$.
$R'_2$ contains $\lambda : a^7 \to (a^6, out)$.

Similar to case (i) above, one can show that to the derivation in $G$ there corresponds the following sequence of transitions in $cN_3'$ starting from the initial configuration $(7,7) : (7,7) \Rightarrow^b (8,7) \Rightarrow^b (8,7) \Rightarrow^c (9,7) \Rightarrow^c (7,7) \Rightarrow^b (6,6)$.

From the above constructions it follows that $L_2$ is the language of control words associated with all label restricted halting computations of $cN_3$ and $cN_3'$.

**Theorem 2.** *(i)* $LLcSNP_4(\{out, in_j\}) \setminus CF \neq \phi$;
*(ii)* $LLcSNP_3(\{here, out, in\}) \setminus CF \neq \phi$.

*Proof.* Consider the non-context-free language $L_3 = \{b^n c^n d^{n+2} | n \geq 1\}$ over the alphabet $\{b, c, d\}$. We build the following LcSN P systems.

(i) $cN_4 = (\Pi_4, \Sigma, \mathcal{C}, lab)$, where $\Pi_4 = (O, \mu, n_1, n_2, n_3, n_4, R_1, R_2, R_3, R_4)$, $\Sigma = \{b, c, d\}$, $\mathcal{C} = \{(4,0,0,0)\}$, $O = \{a\}$, $\mu = [_1[_2[_3]_3]_2[_4]_4]_1$, $n_1 = 3, n_2 = 0, n_3 = 0, n_4 = 3$.
The sets of rules are
$R_1 = \{\ b : a^3 \to (a^3, in_2)(a^3, in_4),\ b : a^3/a^2 \to (a^2, in_2),\ d : a^5/a \to (a, out)\}$,
$R_2 = \{\ c : a^2(a^3)^+/a^3 \to (a^2, in_3),\ c : a^2 \to (a, in_3),\ d : a \to (a, out)\}$,
$R_3 = \{\ d : a(a^2)^+/a^2 \to (a, out),\ d : a \to (a, out)\}$,
$R_4 = \{\ \lambda : a^3 \to (a^3, out)\}$.

Any computation starts from $(n_1, n_2, n_3, n_4) = (3, 0, 0, 3)$. Firstly, the rules $b : a^3 \to (a^3, in_2)(a^3, in_4) \in R_1$ and $\lambda : a^3 \to (a^3, out) \in R_4$ are applied in parallel $n \geq 0$ times and the configuration $(3, 3n, 0, 3)$ is obtained. Then the rule $b : a^3/a^2 \to (a^2, in_2) \in R_1$ and the same rule from $R_4$ are applied in parallel obtaining the configuration $(4, 3n + 2, 0, 0)$. These $n + 1$ transitions are labeled with $b$. Starting with the last configuration only the rule $c : a^2(a^3)^+/a^3 \to (a^2, in_3) \in R_2$ is applicable $n \geq 0$ times and then $c : a^2 \to (a, in_3) \in R_2$ leading to the configuration $(4, 0, 2n + 1, 0)$. All $n + 1$ rules applied have the same label $c$. Now, the rule $d : a(a^2)^+/a^2 \to (a, out) \in R_3$ is applied $n \geq 0$ times and then $d : a \to (a, out)\} \in R_3$, all with the label $d$. One can observe that after the first step of using $d : a(a^2)^+/a^2 \to (a, out) \in R_3$ the configuration $(4, 1, 2n - 1, 0)$ is obtained (or $(4, 1, 0, 0)$, when $n = 0$). After one more step, if $n > 1$, the configuration becomes $(5, 1, 2n - 3, 0)$ and the rule $d : a^5/a \to (a, out)\} \in R_1$ becomes also available. (When $n = 0, 1$, the configurations are $(5, 0, 0, 0)$ and $(5, 1, 0, 0)$, respectively.) After receiving 5 spikes in membrane 1 the rule $d : a^5/a \to (a, out)\} \in R_1$ will start being applied. One can easily observe that after all the spikes from membrane 3 are sent out in $n + 1$ steps, $n \geq 0$, two more transitions labeled $d$ are executed leading to the final configuration $(4, 0, 0, 0)$. Hence, $L(cN_4) = L_3$.

(ii) $cN_4' = (\Pi_4', \Sigma', \mathcal{C}', lab')$, where $\Pi_4' = (O', \mu', n_1', n_2', n_3', R_1', R_2', R_3')$, $\Sigma = \Sigma$, $O' = O$, $\mathcal{C}' = \{(1,0,0)\}$, $\mu = [_1[_2[_3]_3]_2]_1$, $n_1' = 3, n_2' = 0, n_3' = 0$.
The sets of rules are
$R_1' = \{\ b : a^3 \to (a^3, in)(a^3, here),\ b : a^3/a^2 \to (a^2, in),\ d : a^2/a \to (a, out)\}$,
$R_2'$ is obtained from $R_2$ (in $cN_4$) by replacing $in_3$ with $in$ and $R_3' = R_3$.

One can observe that $cN_4'$ is very similar to $cN_4$. The rule $b : a^3 \to (a^3, in_2)(a^3, in_4) \in R_1$ is replaced by $b : a^3 \to (a^3, in)(a^3, here) \in R_1'$ and latter produces what the former together with $\lambda : a^3 \to (a^3, out) \in R_4$ do, apart from the

fact that when $b : a^3/a^2 \to (a^2, in) \in R_2'$ is applied there will be left in membrane 2 only one spike instead of 4 as for $cN_5$. This explains why the rules $d : a^5/a \to (a, out)\} \in R_2$ and $d : a^2/a \to (a, out)\} \in R_2'$ have different regular expressions. Similar to case (i) one can show that $L(cN_4') = L_3$.

□

## 4 Applications

Two applications of control languages associated with cell-like SN P systems to define chain-code picture languages and model work-flow processes are presented.

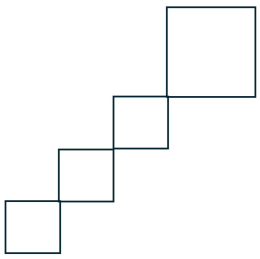
### 4.1 Chain-code picture languages

Array languages have been a special challenge for formal generating devices, mainly devised for string languages. Unconventional models, inspired by biological phenomena, have also met this challenge, and, in the case of P systems, very early (see [9]). A special class of arrays, describable by strings, are those 2D shapes which are composed of only horizontal and vertical lines. In [15] a codification was introduced by Maurer, using the alphabet $V = \{u, d, r, l\}$ where the four symbols stand each for a unit line in a 2D grid, having directions *up*, *down*, *right*, *left* respectively. Families of such pictures are called *chain-code languages*, and finding generative devices for them has been the object of many studies, such as [10] and [11]. Several types of P systems have also been proposed for their generation, such as P systems with parallel rewriting [7], alphabetic flat splicing P systems [13] and three classes of modified alphabetic splicing P systems [8], and, more recently, P colonies which generate strings [5] and labeled SN P systems with mute rules [23].

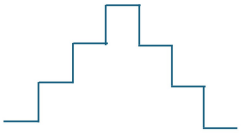
Generation of chain-code pictures using SN P systems with several types of spikes has been done in [6]. In that paper it has been shown that the languages of chain-code representations of $(i)$ a sequence of "kites" and $(ii)$ "double stairs" (Fig. 2) of equal lengths are homomorphic images of string languages of SN P systems with several types of spikes. Here we show that a single LcSN P system can be constructed for generating these two languages. We use the standard chain-code representation using the symbols $l, r, u, d$ for a unit move from a point in the plane to the left, right, up, down directions respectively.

We consider the context-free language $L_c = \{b^n e c^n \mid n \geq 0\}$ over the alphabet $b, c, e$ and the morphisms $h$ and $g$ given by $h(b) = ur, h(e) = u^2r^2d^2l^2, h(c) = dl$, $g(b) = ru, g(e) = r, g(c) = dr$. The construction of a LcSN P system $cN_5$ to generate $L_c$ is similar to $cN_2'$ provided in the proof of Theorem 1. $cN_5 = (\Pi_5, \Sigma, \mathcal{C}, lab)$ where $\Pi_5 = (O, \mu, n_1, n_2, R_1, R_2)$, $\Sigma = \{b, c, e\}$, $\mathcal{C} = \{(0, 0)\}$, $O = \{a\}$, $n_1 = 0, n_2 = 1$. The sets $R_1, R_2$ contain the labeled rules of the membranes 1 and 2 shown in Fig. 3.

It can be seen that the LcSN P $cN_5$ generates $L_c$. Hence,
$h(L_c) = \{(ur)^n u^2r^2d^2l^2(dl)^n \mid n \geq 0\}$ and $g(L_c) = \{(ru)^n r(dr)^n \mid n \geq 0\}$.



A picture in the form of a "kite" represented by the chain code word $(ur)^3u^2r^2d^2l^2(dl)^3$



A picture in the form of "double stairs" represented by the chain code word $(ru)^3r(dr)^3$

**Fig. 2.** Kite and double stairs languages

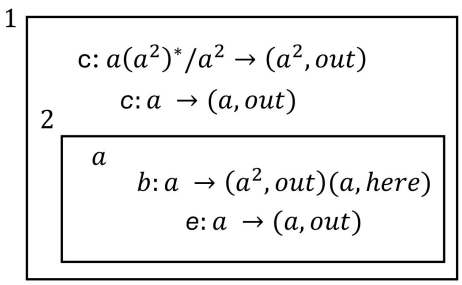



**Fig. 3.** LcSN P system $cN_5$ computing $\{b^n ec^n \mid n \geq 0\}$

The chain-code words in $h(L_c)$ represent the chain-code pictures, referred to as "kites" in [6]. A "kite" is composed of a "tail" - a sequence of unit squares - and a "head" - a $2 \times 2$ square. The sequence $(ur)^n = h(b^n)$ describes the upper part of the "tail", $u^2r^2d^2l^2 = h(e)$ is the "head", and $(dl)^n = h(c^n)$ completes the lower parts of the squares composing the tail. Similarly, $g(L_c)$ is the language of "double stairs" in [6]. While $(ru)^n = g(b^n)$ is the "ascending" stair, $r = g(e)$ is the "platform", and $(dr)^n = g(c^n)$ is the "descending" stair. These are illustrated in Fig. 2 for $n = 3$.

### 4.2 Control-flow modeling

Control-flow modeling is fundamental in workflow management systems, business process modeling, software engineering, and distributed computing. Traditional models such as finite automata, Petri nets, and workflow graphs represent the sequencing of activities and decision-making processes.

The workflow model has been considered in [1], where methods and applications have been also presented. Workflow process models identification from data logs are reported in [2] and flexible and hierarchical decomposition of workflow models is presented in [12].

Membrane computing operations such as division, dissolution and endocytosis, are used in [25] to define a specific cell-like membrane computing model to instantiate workflow patterns and describe their enactment where in a stepwise manner the execution starts from the innermost membrane and stops at the top level, after the previous layers are dissolved. In [4] is considered a different membrane computing approach for specifying workflow patterns, which is based on generalized communicating P systems. The rules of such P systems have the form $(a, i)(b, j) \to (a, k)(b, l)$, where $a, b$ are symbols from a given alphabet with $0 \leq i, j, k, l \leq n$, and 0 denoting the environment and $n$ the number of cells. The model derived from these generalized communicating P systems is called fine-tuned communicating P system and its main feature consists in associating a specific semantics with respect to the number of objects used by each communicating rule. The model is built according to two criteria: (i) all the relevant workflow patterns must be expressible by it; and (ii) the system must provide a solution for creating complex workflow structures by combining the existing patterns. In [24] 20 designs are constructed using spiking neural P systems for 20 classical workflow patterns. The functioning processes that operate both sequentially and simultaneously in the workflow pattern are able to be modeled and simulated in this context. In [3] a model close to spiking neural P systems, called virus machine, is used to specify workflow patterns.

In this section, we propose a novel control-flow modeling framework based on the control languages associated with LcSN P systems which is a cell-like model, but simpler than the model provided in [25]. In the proposed approach, activities are represented by labels attached to spiking rules, while the control words correspond to valid execution traces of the modeled process. The hierarchical membrane structure naturally captures nested workflows, parallel execution, synchronization, and resource movement. We formally define the mapping between several workflow constructs and LcSN P system components. The resulting control language provides a rigorous representation of all valid execution sequences of the process.

This observation suggests a natural interpretation: control words correspond to workflow execution traces. Consequently, LcSN P systems can serve as control-flow modeling tools.

#### 4.2.1. Modeling workflow constructs using LcSN P systems

In this paper, we interpret:

- **Labels** as workflow activities,
- **Configurations** as process states,
- **Transitions** as activity executions,
- **Accepted control words** as valid workflow traces.

Thus, the language $L(cN)$ represents the complete set of legal execution traces (paths) of the modeled process.

The correspondence between workflow concepts and components of a LcSN P system enables us to view the system as a formal control-flow model.

**Table 1.** Workflow Concepts as LcSN P System Components

| **Workflow Concept** | **LcSN P System Component** |
|---|---|
| Activity | Rule Label |
| Process State | Configuration |
| Task Execution | Label-Restricted Transition |
| Workflow Trace | Control Word |
| Parallel Branch | Multiple Membranes |
| Synchronization | Spike Availability |
| Completion State | Final Configuration |

Under this interpretation, each workflow activity is represented by a rule label, and the execution of an activity corresponds to a label-restricted transition between configurations. The distribution of spikes among membranes at a given moment captures the state of the workflow, while a sequence of executed labels forms a control word that records the execution history of a process instance.

Parallelism can be naturally modeled using multiple membranes operating simultaneously, whereas synchronization is achieved through the availability of spikes required for rule application. A workflow execution terminates when the system reaches a designated final configuration.

Consequently, the control language accepted by a LcSN P system represents precisely the set of all valid workflow executions. Formally, if $cN$ is a LcSN P system modeling a workflow, then

$$L(cN) = \{\, w \in \Sigma^* \mid w \text{ corresponds to a valid execution of the workflow} \,\}.$$

Therefore, workflow verification can be reduced to language-theoretic analysis, where properties such as reachability, correctness, deadlock-freeness, and compliance can be studied through the control language of the underlying LcSN P system. Four workflow patterns are modeled below: sequential, choice, loop and parallel constructs.

#### 4.2.2. Sequential construct

The *sequential construct* models a workflow in which activities must be executed in a fixed order. In the workflow

$$A \to B \to C,$$

activity $A$ must be completed before activity $B$ can begin, and activity $B$ must complete before activity $C$ can execute. This execution dependency is naturally captured by the movement of spikes between membranes, as in the LcSN P system $cN_{Seq} = (\Pi_{Seq}, \Sigma, \mathcal{C}, lab)$, where $\Pi_{Seq} = (O, \mu, n_1, n_2, R_1, R_2)$, $\Sigma = \{A, B, C\}$, $\mathcal{C} = \{(0, 0)\}$, $O = \{a\}$, $\mu = [_1[_2]_2]_1$, $n_1 = 2, n_2 = 0$.
The rule sets are:
$R_1 = \{A : a^2 \to (a, in), C : a \to (a, out)\}$,
$R_2 = \{B : a \to (a, out)\}$.

Initially, two spikes are placed in membrane 1, representing that activity $A$ is enabled. The rule $A : a^2 \to (a, in)$ fires when the control symbol $A$ is read. A spike is transferred to membrane 2, indicating the completion of activity $A$ and enabling activity $B$. Once the spike reaches membrane 2, the rule $B : a \to (a, out)$ becomes applicable. The process continues up until the spike reaches membrane 1 again, the symbol C is read and the rule $C : a \to (a, out)$ removes the spike from the membrane and $cN_{Seq}$ reaches the final configuration.

The sequence of configurations during the execution is

$$(2, 0) \Rightarrow^A (0, 1) \Rightarrow^B (1, 0) \Rightarrow^C (0, 0)$$

and the accepted control word is therefore $ABC$, which records the exact order of executing these activities. Any deviation from this order, such as attempting to execute $B$ before $A$ or $C$ before $B$, results in no applicable rule and the computation cannot proceed. Consequently, the accepted language of the system is $L(cN_{Seq}) = \{ABC\}$ which precisely characterizes the valid execution trace of the sequential workflow.

#### 4.2.3. Choice construct

The *choice construct* models a decision point in a workflow where, after completing an activity, exactly one among several alternative activities is executed. In the workflow

$$A \to (B \mid C) \to D,$$

activity $A$ is executed first. After $A$, the process may proceed either through activity $B$ or through activity $C$, but not through both simultaneously. Regardless of the selected branch, the workflow eventually reaches activity $D$.

We construct a LcSN P system $cN_{Ch}$, where $\Sigma = \{A, B, C, D\}$, and $\Pi_{Ch}$ has three membranes, with $\mu = [_1[_2]_2[_3]_3]_1$, $n_1 = 2, n_2 = n_3 = 0$.
The rule sets are:
$R_1 = \{A : a^2 \to (a, in_2), A : a^2 \to (a, in_3), D : a \to (a, out)\}$,
$R_2 = \{B : a \to (a, out)\}$, $R_3 = \{C : a \to (a, out)\}$.

In the LcSN P system presented above when two spikes are available, only one of the rules $A : a^2 \to (a, in_2)$, $A : a^2 \to (a, in_3)$ can be applied during a transition. The application of the first rule sends a spike to membrane 2,

thereby selecting branch $B$. The application of the second rule sends the spike to membrane 3, i.e., selecting branch $C$. Thus, the non-deterministic selection of a rule naturally models the workflow decision. Each of the membranes 2 and 3 has a rule labeled $B$ or $C$, respectively, corresponding to these activities and sending out to membrane 1 a spike. Finally, the rule $D : a \to (a, out)$ is executed, as a continuation of either activity $B$ or $C$. The rules corresponding to activities $A$ and $D$ are all located in membrane 1. Given that initially there are two spikes in membrane 1 ($n_1 = 2$) and using two spikes to activate the rules corresponding to activity $A$ and only one for the rule corresponding to $D$ one prevents the execution of the rule labeled with $D$ before any of those labeled with $A$. Hence, the two controlling words are $ABD$ and $ACD$, corresponding to the two branches of the choice construct.

The control language is $L(cN_{Ch}) = \{ABD, ACD\}$. Consequently, the control language precisely captures all possible paths through the decision point, making the LcSN P system a natural formal model for workflow choice constructs.

#### 4.2.4. Loop construct

The *loop construct* models repetitive execution in a workflow, where a set of activities may be executed multiple times until a termination condition is satisfied. Consider the approval process

$$A(RA)^*F,$$

where $A$ denotes an approval activity, $R$ denotes a revision activity, and $F$ denotes finalization. After an approval, the process may either be finalized immediately or returned for revision. Following a revision, the document must be approved again. This cycle may repeat any number of times before the process is finally completed.

We construct a LcSN P system $cN_{Loop}$, where $\Sigma = \{A, R, F\}$, $\mathcal{C} = \{(0, 0)\}$, and $\Pi_{Loop}$ has two membranes, with $\mu = [_1[_2]_2]_1$, $n_1 = 2, n_2 = 0$.
The rule sets are:
$R_1 = \{A : a^2 \to (a^2, in), A : a^2 \to (a, here), F : a \to (a, out)\}$,
$R_2 = \{R : a^2 \to (a^2, out)\}$.

Membrane 1 represents the approval activity $A$ and the finalization activity $F$, while membrane 2 represents the revision activity $R$.

Initially, two spikes are present in membrane 1. Either $A : a^2 \to (a^2, in)$ or $A : a^2 \to (a, here)$ is selected; both rules correspond to the execution of an approval activity. The first choice will lead to a revision activity, whereas the second will get to a finalization. Indeed, in the first case, the two spikes are transferred to membrane 2, where $R : a^2 \to (a^2, out)$ models a revision request. When this rule is applied, the two spikes are sent back to membrane 1, forcing another approval cycle. Consequently, the sequence of activities $RA$ can be repeated arbitrarily many times. When $A : a^2 \to (a, here)$ is selected to be applied, this signifies that the approval activity will be followed by a finalization.

Indeed, it produces a spike which makes the rule $F : a \to (a, out)$, which models successful completion of the approval process, applicable. Consequently, this will end the loop workflow.

Every execution consists of an initial approval, followed by zero or more repetitions of the revision–approval cycle, and finally a finalization step. Therefore, the control language accepted by the system is $L(cN_{Loop}) = \{A(RA)^n F | n \geq 0\}$.

### 4.2.5. Parallel construct

The *parallel construct* models situations in which multiple activities are initiated simultaneously and may proceed independently before synchronizing at a later stage. Consider the workflow in which activity $A$ initiates two concurrent branches $B$ and $C$, both of which must complete before activity $D$ can execute.

We construct a LcSN P system $cN_{Par}$, where $\Sigma = \{A, B, C, D\}$, and $\Pi_{Par}$ has three membranes, with $\mu = [_1[_2]_2[_3]_3]_1$, $n_1 = 1, n_2 = n_3 = 0$.
The rule sets are:
$R_1 = \{A : a \to (a, in_2)(a, in_3), D : a^2 \to (a, out)\}$,
$R_2 = \{B : a \to (a, out)\}$,
$R_3 = \{C : a \to out\}$.

In the LcSN P model $cN_{Par}$ membrane 1 represents the activities $A$ and $D$. The rule $A : a \to (a, in_2)(a, in_3)$ consumes one spike and sends one spike to membrane 2 and another to membrane 3. This rule models the parallel split of the workflow. After the execution of $A$, both branches become active simultaneously. Membrane 2 represents activity $B$ and contains the rule $B : a \to (a, out)$ while membrane 3 represents activity $C$ and contains the rule $C : a \to (a, out)$. The execution of these rules corresponds to the completion of activities $B$ and $C$, respectively. Since the two branches are independent, their executions may occur in any order. Each branch forwards a spike to membrane 1 upon completion. In membrane 1 takes place the synchronization followed by activity $D$, given by the rule $D : a^2 \to (a, out)$. This rule requires two spikes for execution. Consequently, activity $D$ cannot be executed until both branches $B$ and $C$ have completed and delivered their spikes. The requirement of two spikes naturally models synchronization or the AND-join construct commonly found in workflow systems.

Once both spikes are available, the rule labeled $D$ terminates the computation. This corresponds to successful completion of the workflow.

Since activities $B$ and $C$ execute concurrently, their relative ordering is not fixed. The control language therefore contains all valid interleavings that preserve the causal dependencies imposed by the workflow.

Typical accepted control words are $ABCD$ and $ACBD$.

Hence, the accepted control language can be expressed as
$L(cN_{Par}) = \{ABCD,\ ACBD\}$.

Thus, the LcSN P system captures both parallel execution and synchronization using the natural mechanisms of spike distribution and spike accumulation.

The rule labeled $D$ requiring two spikes acts as an implicit synchronization barrier, ensuring that activity $D$ is executed only after all parallel branches have completed.

Finally, a **unified workflow representation** is provided in Table 2. It captures different workflow artifacts and their corresponding LcSN P system features and control language elements.

**Table 2.** Workflow constructs and their LcSN P representations.

| Workflow Pattern | LcSN P Feature | Control Language |
|---|---|---|
| Sequence | Spike Passing | $ABC$ |
| Choice | Non-deterministic Rules | $ABD + ACD$ |
| Loop | Cyclic Communication | $A(RA)^*F$ |
| Parallelism | Multiple Membranes | Interleavings |
| Synchronization | Spike Accumulation | Join Point |
| Completion | Final Configuration | Accepted Word |

## 5 Conclusions and Further Work

In this paper control languages associated to two classes of cell-like spiking neural P systems have been investigated by considering their relationships with regular and context-free languages. Further work is aimed at investigating the relationships of these families of languages with context-sensitive and recursively enumerable language families as well as extending the association of control languages to other generative mechanisms - the virus machine model [16], very close to the membrane computing paradigm, is a potential candidate.

This work introduces a novel interpretation of labeled cell-like spiking neural P systems as control-flow modeling tools. By associating workflow activities with rule labels, accepted control words become workflow execution traces. The resulting framework combines the expressive power of membrane computing with formal language representations of workflows. Future work includes modeling of other control-flow patterns workflow verification, conformance checking, process mining, and resource-aware control-flow analysis using control languages of cell-like SN P systems.

**Ackowledgement.** The authors would like to thank the anonymous reviewers for their comments, observations and suggestions that significantly helped improving the quality and content of the paper.

## References

1. van der Aalst, W., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press (2004)

2. van der Aalst, W., Weijters, A.J.M.M., Măruşter, L.: Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering **16**(9), 1128–1142 (2004)
3. de Arellano, A.R., Cabarle, F.G.C., Orellano-Martín, D., Riscos-Núñez, A., Pérez-Jiménez, M.J.: Virus machines at work: Computations of workflow patterns. In: DSA ISC (1). pp. 208–219 (2024)
4. Balaskó, Á.: On a workflow model based on generalized communicating P systems. Computer Sci. J. Moldova **17**(1), 45–68 (2016)
5. Bera, S., Ceterchi, R., Mathew, L., Subramanian, K.G.: String generating P colonies with Eilenberg machines (submitted)
6. Ceon, Y.P., Anandharaj, H.C., Jebasingh, S., Chandy, D.A.: Generation of chain code pictures using cell-like spiking neural P system with several types of spikes. J. Membr. Comput. **4**(3), 243–250 (2022)
7. Ceterchi, R., Subramanian, K.G., Venkat, I.: P systems with parallel rewriting for chain code picture languages. In: Beckmann, A. et al. (ed.) Evolving Computability, CiE 2015. LNCS, vol. 9136, pp. 145–155. Springer (2015)
8. Ceterchi, R., Gheorghe, M., Kuppusamy, L., Subramanian, K.G.: Three classes of modified alphabetic flat splicing P systems. J. Membr. Comput. **7**(3), 296–311 (2025)
9. Ceterchi, R., Mutyam, M., Păun, Gh., Subramanian, K.G.: Array-rewriting P systems. Nat. Comput. **2**(3), 229–249 (2003)
10. Dassow, J., Habel, A., Taubenberger, S.: Chain-code pictures and collages generated by hyperedge replacement. In: Cuny, J. et al. (ed.) Graph Grammars 1994. Lecture Notes in Computer Science, vol. 1073, pp. 412–427. Springer (1994)
11. Drewes, F.: Some remarks on the generative power of collage grammars and chain-code grammars. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Theory and Application of Graph Transformations. pp. 1–14. Springer Berlin Heidelberg (2000)
12. Huang, T.H., Jansen, L.M., Pegoraro, M., Park, G., van der Aalst, W.: Flexible and hierarchical decomposition of workflow nets for process analysis. In: Lecture Notes in Computer Science Advanced Information Systems Engineering. pp. 305–321. Springer Nature Switzerland (2026)
13. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae **71**(2–3), 279–308 (2006)
14. Krithivasan, K., Păun, Gh., Ramanujan, A.: On controlled P systems. Fundamenta Informaticae **131**(3–4), 451–464 (2014)
15. Maurer, H.A., Rozenberg, G., Welzl, E.: Using string languages to describe picture languages. Information and Control **54**, 155–185 (1982)
16. Orellana-Martín, D., Ramírez-de-Arellano, A., Pérez-Jiménez, M.J.: Simulating and validating virus machines. Nat. Comput. **24**(2), 311–320 (2025)
17. Păun, Gh.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693, https://doi.org/10.1006/jcss.1999.1693
18. Păun, Gh.: Membrane Computing - An Introduction. Springer, Verlag (2002)
19. Păun, Gh., Pérez-Jiménez, M.: Languages and P systems: recent developments. Computer Sci. J. Moldova **20**(2), 112–132 (2014)
20. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
21. Ramanujan, A., Krithivasan, K.: Control languages associated with spiking neural P systems. Romanian J. Inf. Sci. Tech. **15**(4), 301–318 (2012)

22. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Vol. 1: Word, Language, Grammar. Springer-Verlag New York, Inc., New York, NY, USA (1997)
23. Sebastian, J., Mathew, L.: Generation of chain code pictures using labeled SN P systems with mute rules. Parallel Processing Letters (accepted)
24. Song, T., Zeng, X., Zheng, P., Jiang, M., Rodríguez-Patón, A.: A parallel workflow pattern modeling using spiking neural P systems with colored spikes. IEEE Transactions on Nanobioscience **17**(4), 474–484 (2018)
25. Verma, R., Ahmed, T., Srivastava, A.: Expressing workflow and workflow enactment using P systems. In: Gheorghe, M., Sosík, P., Vavrečková, Š. (eds.) The $15^{th}$ International Conference on Membrane Computing. pp. 345–359. Institute of Computer Science, Faculty of Philosophy and Science, Silesian University in Opava (2014)
26. Wu, T., Zhang, Z., Păun, Gh., Pan, L.: Cell-like spiking neural P systems. Theoretical Computer Science **623**, 180–189 (2006)

# Spiking Neural P Systems in kPWorkbench

Ștefan Iacob[1],
Florentin Ipate[1],
Marian Gheorghe[2], and
Ionuț Mihai Niculescu[1]

[1]University of Bucharest, Faculty of Mathematics and Computer Science,
14 Academiei St, 010014 Bucharest, Romania.
[2]School of Computing and Engineering, University of Bradford,
West Yorkshire, Bradford BD7 1DP, UK.

**Abstract.** In this paper, we introduce a unified translation framework within kPWorkbench to facilitate the modeling and simulation of advanced Spiking Neural P systems. Our approach enables the integration of models from disparate simulators by natively mapping standard SN P systems, alongside variants utilizing anti-spikes and synaptic weights, into executable kP-Lingua constructs. We also illustrate our approach through several examples, including sorting networks and NP-complete problem solvers. While our current implementation does not yet support synaptic delays, we outline a formal state-machine mapping for future integration.

**Keywords:** Membrane Systems · kernel P Systems · Spiking Neural P Systems

## 1 Introduction

Membrane computing, a research field initiated by Gh. Păun, is a computational paradigm inspired by the biological structure and biochemical interactions of living cells. The field has rapidly expanded across diverse research directions, resulting in the formalization of numerous classes of membrane systems, broadly categorized into cell-like, tissue-like, and neural-like architectures. Among the neural-like models, Spiking Neural P systems (SN P systems) have emerged as a highly prominent class. Introduced in [31], these systems abstract the neurophysiological behavior of brain neurons, which process and communicate information by transmitting electrical impulses along axons. Since their inception, standard SN P systems have been rigorously investigated, establishing themselves as a powerful, Turing-complete computational framework governed by a foundational mechanism of spiking and forgetting rules.

Significant results on SN P systems have been reported in the literature, leading to a massive expansion of the original model. A multitude of SN P system variants have been considered and investigated with features such as: anti-spikes and inhibitory synapses [18], inhibitory rules [28], request rules [35],

multiple channels [30], extended channel rules [15], colored spikes [37], white hole neurons [34], astrocytes [20], polarizations [41], lateral inhibition [14], rules on synapses [36], weights and potential thresholds [39], dynamic thresholds [29], weights on synapses and extended rules with delay [21], dendrites [23], plasticity [1,16], neuron division and budding [17], and scheduled synapses [2]. Furthermore, entirely new classes of systems have emerged, such as axon P systems [44], fuzzy reasoning SN P systems [27], coupled SN P systems [25], numerical SN P systems [40], and nonlinear SN P systems [24].

Various research topics based on SN P system models have been considered, supported by comprehensive bibliographies [19], surveys of the main classes of SN P systems [12], and overviews of learning SN P systems [3]. Theoretical results and applications of SN P systems are widely discussed in the literature [33], culminating in two major research textbooks on SN P systems published in 2024 [26, 42].

For several of the above-mentioned SN P systems, different execution strategies for applying rules, as well as distinct methods for encoding the results of the computation, have been extensively investigated [12]. Among these many variants, standard SN P systems provide the theoretical baseline, while SN P systems with weights and SN P systems with anti-spikes represent two of the most heavily utilized extensions. Weighted SN P systems assign numerical values to synapses to reflect their transmission strength. By introducing this variable transmission efficacy, the topology of the system closely mirrors the architecture of classic artificial neural networks (ANNs) and third-generation Spiking Neural Networks (SNNs). Crucially, the presence of synaptic weights allows for the introduction of synaptic plasticity into the membrane computing framework. Rather than acting as purely static computational devices, these weights can be dynamically adjusted using established training algorithms, such as Hebbian learning rules or adaptations of backpropagation. This capability has led to the emergence of "learning SN P systems," which have become highly prevalent in machine learning models, enabling the system to perform complex neuro-classification tasks, pattern recognition, and fault diagnosis [22, 43]. Concurrently, SN P systems with anti-spikes introduce inhibitory objects that mutually annihilate with regular spikes, providing critical control and synchronization mechanisms.

As the theoretical study has advanced, various software tools, such as UPSimulator [7], the P-Lingua/MeCoSim ecosystem [32], and Snapse [4], have been developed to simulate their behavior. However, the current landscape of these simulators is highly fragmented. Because each tool relies on unique domain-specific languages and internal architectures, models designed in one framework cannot be easily exported, simulated, or formally verified in another.

To bridge this gap, kernel P (kP) systems were introduced to capture the most utilized features of existing membrane computing models in a unified manner, while integrating advanced concepts such as compartment types, guarded rules governed by Boolean conditions, and user-defined execution strategies. This expressive capability makes the kP system model highly amenable to describing complex problems across diverse domains, ranging from computer science

topics like communication and synchronization to applications in synthetic biology [9,10]. Conceived as a comprehensive modeling paradigm, kP systems allow researchers to formally specify problems, verify model correctness, and rigorously test implementations. This theoretical framework is fully supported by an expressive domain-specific formal language and executed via kPWorkbench [5], a robust software suite that provides dedicated modules for both the simulation and formal verification of kP systems. A detailed presentation of this tool was made in [11].

This paper represents a continuation of the ongoing efforts to expand the modeling capabilities of kPWorkbench, specifically building upon the recently developed indexation mechanisms [8] and the foundational translation concepts introduced in [6]. By extending these ideas, we investigate the formal relationship between advanced SN P systems and kP systems. We are proposing a unified simulation and translation framework integrated directly into kPWorkbench, demonstrating how standard SN P systems, alongside those utilizing weights and anti-spikes, can be natively mapped into kP-Lingua constructs. By rigorously preserving the operational semantics of these variants, this extension allows kPWorkbench to seamlessly ingest, parse, and verify models originating from disparate tools such as UPSimulator, P-Lingua, and Snapse.

The paper is organized as follows. Section 2 briefly recalls the definitions of kP systems and the investigated SN P system variants. Section 3 details the formal mapping and translation mechanism required to preserve the semantic behavior of these systems within kPWorkbench. In Section 4, a series of problems, including sorting networks, NP-complete problem solvers, and arithmetic modules, are presented to validate the translation pipeline across multiple simulators. Section 5 discusses related works and provides a comparison with existing simulation frameworks. Finally, conclusions and directions for future work are drawn in Section 6, including a discussion on the current limitations of the framework regarding synaptic and firing delays.

## 2 kPWorkbench and Spiking Neural P Systems

In this section, we briefly recall the formal definitions and basic concepts of Spiking Neural P (SN P) systems and kernel P (kP) systems, along with the specific variants investigated in this work.

### 2.1 Standard Spiking Neural P Systems

Spiking Neural P systems are a class of distributed and parallel computing models inspired by the neurophysiological behavior of biological neurons. A standard SN P system of degree $m \geq 1$ is a construct of the form:

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$$

where:

- $O = \{a\}$ is the singleton alphabet (the object $a$ is called a spike).
- $\sigma_1, \ldots, \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i)$, with $1 \leq i \leq m$, where:
  - $n_i \geq 0$ is the initial number of spikes contained in $\sigma_i$.
  - $R_i$ is a finite set of rules of the following two forms:
    1. *Spiking rules:* $E/a^c \to a; d$, where $E$ is a regular expression over $O$, $c \geq 1$, and $d \geq 0$ represents the delay. If the neuron $\sigma_i$ contains $k$ spikes such that $a^k \in L(E)$ and $k \geq c$, the rule can be applied. Upon application, $c$ spikes are consumed immediately. The neuron becomes *closed* (inactive) for $d$ steps. After $d$ steps, the neuron becomes *open* again and emits exactly one spike to all neurons $\sigma_j$ such that $(i, j) \in syn$.
    2. *Forgetting rules:* $a^s \to \lambda$, with $s \geq 1$, and $a^s \notin L(E)$ for any spiking rule in $R_i$. If exactly $s$ spikes are present, they are removed from the neuron.
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$ represents the synapses between neurons, with $(i, i) \notin syn$ for $1 \leq i \leq m$.
- $in, out \in \{1, \ldots, m\}$ indicate the input and output neurons, respectively.

If multiple rules in $R_i$ are applicable simultaneously, only one is chosen nondeterministically.

### 2.2 SN P Systems with Anti-spikes

In biological neural networks, neurons process both excitatory and inhibitory signals. To formally capture this duality, SN P systems with anti-spikes extend the classic model by introducing a secondary object, the anti-spike, denoted by $\bar{a}$.

Formally, an SN P system with anti-spikes preserves the general architectural topology of the standard model (*syn*, *in*, and *out*) but modifies the alphabet and the internal neuron definitions to accommodate the new object. The distinct extensions are defined as follows:

- *Alphabet:* $O = \{a, \bar{a}\}$, where $a$ represents a spike and $\bar{a}$ represents an anti-spike.
- *Neurons:* Each neuron $\sigma_i = (n_i, R_i)$ utilizes $n_i$ to represent the initial number of objects (which must be exclusively spikes or exclusively anti-spikes). The rule set $R_i$ is expanded to include:
  1. *Spiking rules:* $E/b^c \to b'$, where $E$ is a regular expression defined exclusively over $\{a\}$ or over $\{\bar{a}\}$, $b, b' \in \{a, \bar{a}\}$, and $c \geq 1$. This yields four operational categories based on the consumed and produced objects: $(a, a), (a, \bar{a}), (\bar{a}, a)$, and $(\bar{a}, \bar{a})$.
  2. *Forgetting rules:* $b^c \to \lambda$, where $b \in \{a, \bar{a}\}$ and $c \geq 1$.

The fundamental operational characteristic of this variant is that spikes and anti-spikes cannot stay together, and they instantaneously annihilate each other. If a neuron receives objects such that it temporarily contains $r$ spikes and $s$ anti-spikes, an implicit annihilation rule $a\bar{a} \to \lambda$ is immediately applied in a maximal

manner. Consequently, either $r - s$ spikes or $s - r$ anti-spikes remain, depending on whether $r \geq s$ or $s \geq r$.

Crucially, this mutual annihilation takes no time and has strict priority over any spiking or forgetting rules. Thus, a neuron always contains exclusively one type of object before any standard rule evaluation occurs. Additionally, by convention in generative or accepting modes, the designated output neuron *out* is restricted to producing only spikes.

### 2.3 SN P Systems with Weights

Synapses exhibit varying degrees of transmission efficacy, meaning the impact of a spike depends heavily on the specific connection it travels through. To more accurately model this biological reality, and to bridge the gap with the architecture of artificial neural networks, Spiking Neural P (SN P) systems with weights introduce numerical values to the synapses and replace discrete multisets with a continuous or integer-based potential.

Formally, following the model investigated by Wang et al., an SN P system with weights alters the standard construct by redefining the internal composition of the neurons and the properties of the synapses:

- *Neurons:* Each neuron takes the form $\sigma_i = (p_i, R_i)$, where $p_i$ is the initial potential. The finite set of rules $R_i$ consists of spiking rules of the form $T_i/d_s \to 1$. Here, $T_i \geq 1$ is the fixed firing threshold potential of the neuron $\sigma_i$, and $d_s$ is the amount of potential consumed, restricted by $0 < d_s \leq T_i$.
- *Synapses:* The topology is redefined as $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\} \times \mathbb{R}_c$. A synapse is represented by a triplet $(i, j, r)$, where $r \neq 0$ dictates the specific transmission weight of the connection.

The introduction of potentials and weights significantly alters the operational semantics. Rather than evaluating regular expressions, a neuron $\sigma_i$ utilizes its single, fixed threshold $T_i$. The rules are applied as follows:

- If the current potential $p$ equals its threshold ($p = T_i$), all rules in $R_i$ are enabled, and one rule $T_i/d_s \to 1$ is chosen non-deterministically. Executing this rule consumes $d_s$ potential and emits exactly one unit of potential (a spike).
- The emitted unit passes immediately to all connected neurons $\sigma_j$. However, the receiving neuron $\sigma_j$ collects a quantity of potential equal to the synaptic weight $r$, which is added to its existing potential.
- If the potential strictly exceeds the threshold ($p > T_i$), it remains unchanged, and no rule is applied.
- If the potential is strictly less than the threshold ($p < T_i$), the neuron loses its charge and its potential immediately resets to 0.

While the theoretical framework allows potentials and weights to be drawn from real computable numbers ($\mathbb{R}_c$), they are frequently restricted to integers ($\mathbb{Z}$) in practical simulation and computational verification. By utilizing positive and

negative weights, the system naturally supports both excitatory and inhibitory connections, allowing it to achieve Turing universality while offloading complex signal routing logic directly onto the synapses.

Beyond the theoretical implications of Turing universality, the introduction of synaptic weights fundamentally bridges the gap between membrane computing and the functional architectures of classic artificial neural networks (ANNs) and third-generation Spiking Neural Networks (SNNs) [22]. In standard SN P systems, synapses act as static, uniform communication channels. However, by incorporating variable weights $r$, the framework naturally accommodates the concept of *synaptic plasticity*. This transition allows the SN P system to evolve from a rigid, rule-based computational device into a dynamic, learning-capable network.

Because the impact of a spike can be amplified or diminished based on the current value of the synaptic weight, these weights can be iteratively adjusted during computation. This enables the integration of established training mechanisms directly into the membrane computing framework, such as Hebbian learning rules, Spike-Timing-Dependent Plasticity (STDP), or adaptations of backpropagation [3].

Consequently, this weighted architecture has catalyzed the development of learning SN P systems. By dynamically tuning the transmission efficacy between neurons, these extended models have been successfully deployed in a variety of modern machine learning applications. They have demonstrated high accuracy and energy efficiency in pattern recognition, complex neuro-classification tasks, and real-time fault diagnosis [13, 43]. The ability to seamlessly integrate learning algorithms while maintaining the biological plausibility of the P system framework cements weighted SN P systems as a premier model for bio-inspired artificial intelligence.

### 2.4 Kernel P Systems

Kernel P systems (kP systems) have been introduced to formalize and integrate the most successful and widely used features of various P system classes into a single, cohesive, and expressive mathematical framework. This unification facilitates rigorous modeling, simulation, and formal verification.

Formally, a kernel P system of degree $n \geq 1$ is defined as a tuple:

$$k\Pi = (T, \Sigma, C_1, \ldots, C_n, \mu)$$

where the components are defined as follows:

- $T$ is a finite set of compartment types. Each compartment type $t \in T$ is defined as a pair $t = (R, \sigma)$, where $R$ is a finite set of rules and $\sigma$ is an execution strategy.
- $\Sigma$ is a finite alphabet consisting of objects.
- $C_1, \ldots, C_n$ represent the $n$ compartments of the system. Each compartment is of the form $C_i = (t_i, w_i)$, with $1 \leq i \leq n$, where $t_i \in T$ denotes the type of the compartment and $w_i$ is a multiset over $\Sigma$ defining the initial objects contained within $C_i$.

– $\mu$ is the initial topological structure of the system. It is defined as an undirected graph where the vertices correspond to the compartments $C_1, \ldots, C_n$, and the edges denote the available communication links between them.

A defining characteristic of kP systems is the use of guarded rules. A guard $g$ is a Boolean condition evaluated against the current multiset of a compartment. Guards are constructed using multiset evaluations (e.g., relations like $=, \geq, \leq, >, <$) connected by standard propositional logic operators ($\wedge, \vee, \neg$).

The rules within the set $R$ of a compartment type are primarily rewriting and communication rules, typically of the form $x \to y\{g\}$. Here, $x \in \Sigma^+$ is the multiset of objects to be consumed, $y$ is the multiset of produced objects (which may include target indicators for communication), and $g$ is the associated guard. A rule is considered applicable only if its guard $g$ evaluates to true under the current multiset configuration of the compartment.

The execution strategy $\sigma$ defined within a compartment type dictates the precise manner in which the rules of $R$ are applied during a computation step. kPWorkbench natively supports various strategies, including *choice* (where one applicable rule is selected non-deterministically), *maximal parallel* (where applicable rules are applied concurrently to the maximum extent possible), and *sequence* (where blocks of rules are executed in a strict predefined order).

## 3 Formal Mapping and Translation of SN P Systems

### 3.1 Theoretical Foundations of the Mapping

The formal relationship between Spiking Neural P systems and kernel P systems has been extensively investigated, establishing a strict theoretical equivalence [6]. As demonstrated by Gheorghe et al., the process of mapping various classes of SN P systems into kP systems is a constructive one. Because these mappings are constructive, they provide a direct algorithmic foundation for software implementation, allowing complex SN P variants to be simulated entirely within standard kP system constructs without requiring modifications to the core execution engine.

The baseline structural mapping establishes a direct correspondence where neurons are instantiated as compartments, and synapses are translated into a network of communication links. The operational semantics of standard SN P systems are preserved by translating regular expressions into Boolean guards, allowing spiking and forgetting rules to be modeled as standard kP rewriting and communication rules. Because standard SN P systems apply rules non-deterministically, the execution strategy for these compartments is defined as a simple choice.

For advanced variants, the constructive mapping dynamically alters the alphabet and the execution strategies. To prove that the translation framework rigorously preserves the step-by-step operational semantics of the source SN P models, we formalize the correctness of the mapping using strong bisimulation. Two systems are strongly bisimilar if their transition systems match exactly

at every step, meaning every configuration change in the source system is perfectly mirrored by a corresponding configuration change in the target system, guaranteeing operational trace equivalence.

**Theorem 1.** *For any SN P system with anti-spikes, $\Pi_{a\bar{a}}$, let $k\Pi_{a\bar{a}}$ be the kP system generated by the translation mapping. The transition system of $\Pi_{a\bar{a}}$ and the transition system of $k\Pi_{a\bar{a}}$ are strongly bisimilar, guaranteeing operational trace equivalence.*

*Proof.* Let us consider an SN P system with anti-spikes, $\Pi_{a\bar{a}} = (O, \sigma_1, \ldots, \sigma_m, syn, i_0)$, where $O = \{a, \bar{a}\}$. In $\Pi_{a\bar{a}}$, if an anti-spike meets a spike in a given neuron, they annihilate each other instantaneously via an implicit rule $a\bar{a} \to \lambda$. This mutual annihilation takes no time and has strict priority over any spiking or forgetting rules.

Let the configurations of both systems be defined by a vector of multisets representing the objects inside each neuron/compartment. We define a mapping relation $\mathcal{R}$ such that a configuration $C$ in $\Pi_{a\bar{a}}$ is related to a configuration $kC$ in $k\Pi_{a\bar{a}}$ if and only if their multiset vectors are identical.

In the constructed kP system $k\Pi_{a\bar{a}}$, the alphabet is mapped directly to $\{a, \bar{a}\}$. For every rule $r : E/b^c \to b' \in R_i$, a corresponding rule $r' : b^c \to (b', t_{i_1}) \ldots (b', t_{i_{h_i}})\{g_E\}$ is added to the set of rules $R'_i$. Crucially, every set of rules $R'_i$ contains an additional explicit annihilation rule, $r_e : a\bar{a} \to \lambda$. The execution strategy $\delta_i$ for each compartment in $k\Pi_{a\bar{a}}$ is defined as a sequence of two blocks: $\delta_i = \{r_e\}^T \{r'_1, \ldots, r'_{k_i}\}$.

Suppose $\Pi_{a\bar{a}}$ undergoes a transition $C \to C'$. During this step, the instantaneous mutual annihilation occurs first, followed by the non-deterministic application of standard spiking/forgetting rules. In $k\Pi_{a\bar{a}}$, the macroscopic step $\delta_i$ evaluates the first block $\{r_e\}^T$ in a maximal parallel manner, matching the instantaneous $a\bar{a} \to \lambda$ priority of the native SN P system. Because the kP system engine resolves sequential blocks within a single macroscopic transition step, the intermediate state is not exposed, and the subsequent choice block $\{r'_1, \ldots, r'_{k_i}\}$ is evaluated exactly against the annihilated multiset. Thus, $kC \to kC'$ exactly mirrors $C \to C'$, preserving $\mathcal{R}(C', kC')$. Since the transition systems map one-to-one, $\Pi_{a\bar{a}}$ and $k\Pi_{a\bar{a}}$ are strongly bisimilar.

Similarly, for SN P systems with weights, continuous potentials, and thresholds, the operational trace equivalence is maintained by grouping the continuous mathematical evaluations into a sequential multiset evaluation strategy.

**Theorem 2.** *For any SN P system with weights, $\Pi_{w,p,t}$, let $k\Pi_{w,p,t}$ be the kP system generated by the translation mapping. The transition system of $\Pi_{w,p,t}$ and the transition system of $k\Pi_{w,p,t}$ are strongly bisimilar.*

*Proof.* Consider an SN P system $\Pi_{w,p,t}$ where each neuron $\sigma_i$ has a current potential $p$, a threshold $T_i$, and a set of rules $R_i$ of the form $T_i/q_{j,i} \to 1$, where $0 < q_{j,i} \leq T_i$. According to the operational semantics defined by Wang et al., if $p < T_i$, the potential vanishes (is reset to 0). If $p = T_i$, one rule is chosen

non-deterministically, and the neuron fires, sending a spike that adds a specific weight $r$ (positive or negative) to connected neurons.

In the corresponding kP system, $k\Pi_{w,p,t}$, continuous/integer potentials are discretized into multisets of positive objects ($a$) and negative objects ($n$). The execution strategy $\delta_i$ is mapped as a sequence of three blocks: $\delta_i = \{r_{a,n}\}^T\{r_{reset}\}^T \{r_{fire}\}$.

During a transition step in the native system $\Pi_{w,p,t}$, the net potential is calculated, sub-threshold potentials are reset to 0, and applicable rules are executed. In $k\Pi_{w,p,t}$, the first block ($\{r_{a,n}\}^T$) computes the net potential via maximal parallel annihilation ($1a, 1n \to \lambda$). Immediately after, the second block ($\{r_{reset}\}^T$) applies guarded rules (e.g., $< T_i : 1a \to \lambda$) to reset sub-threshold potentials to 0, strictly matching the behavior where potentials lower than the threshold vanish. Finally, the third block arbitrarily selects one of the threshold firing rules.

Because the discrete multiset operations are forcefully grouped into this rigid sequential execution strategy ($\delta_i$), the kP system resolves the net potential calculation, the threshold checking, and the signal emission entirely within one macroscopic transition step. Therefore, every configuration change $C \to C'$ in $\Pi_{w,p,t}$ is perfectly matched by $kC \to kC'$ in $k\Pi_{w,p,t}$, establishing a strong bisimulation and guaranteeing operational trace equivalence between the two systems.

### 3.2 Translation Pipeline and Implementation

Building upon these constructive theoretical foundations, a comprehensive software translation pipeline was implemented within kPWorkbench. The current software landscape for membrane computing relies on highly varied domain-specific languages and serialization formats. To overcome this fragmentation, the proposed framework employs specialized parsing instrumentations tailored to the distinct architectures of the source simulators.

Specifically, models defined in P-Lingua are ingested utilizing an ANTLR4-generated lexer and parser, which constructs an Abstract Syntax Tree (AST) to extract formal algebraic declarations and structural macros. For models originating from UPSimulator, the framework processes the input with targeted regular expression engines designed to identify and extract multiset definitions and object-tunnel routing rules. Finally, graphical models constructed in Snapse are parsed via a structural object mapping mechanism, enabling the extraction of rule arrays, synaptic connections, and the underlying visual-spatial node data inherent to the tool.

Rather than developing independent, hardcoded translators for each of these formats into kP-Lingua, the pipeline abstracts the structural topologies and operational semantics into a universal, intermediate Extensible Markup Language (XML) representation. This XML schema captures the common denominators of all SN P systems: neurons, initial multisets, synapses, and rules (characterized by conditions, consumed objects, produced objects, and delays).

To rigorously define this intermediate architecture, the following Document Type Definition (DTD) outlines the formal structural constraints of the unified XML schema. Furthermore, to illustrate the practical application of this schema,

a fully worked XML instance generated for the 4-channel bitonic sorting network is provided in Appendix A:

```
<!ELEMENT model (neuronType*, neuron*, synapse*)>
<!ELEMENT neuronType (rules)>
<!ATTLIST neuronType id ID #REQUIRED threshold CDATA #IMPLIED>
<!ELEMENT rules (rule*)>
<!ATTLIST rule id ID #REQUIRED regex CDATA #REQUIRED
          consumed CDATA #REQUIRED produced CDATA #REQUIRED
          producedSymbol CDATA "a" delay CDATA "0" type CDATA
          #REQUIRED>
<!ELEMENT neuron EMPTY>
<!ATTLIST neuron id ID #REQUIRED type IDREF #REQUIRED
          initialSpikes CDATA "0">
<!ELEMENT synapse EMPTY>
<!ATTLIST synapse source IDREF #REQUIRED target IDREF #REQUIRED
          weight CDATA "1" loop CDATA #IMPLIED>
```

From this unified XML format, a dynamic generation engine produces the target kP-Lingua code by programmatically applying the constructive mappings defined in the theoretical literature. The engine factors out the general topological mapping, instantiating the compartment multiset distributions and the global link graph. Crucially, the pipeline delegates the variant-specific operational behaviors directly to the compartment type definitions. It translates the XML rule attributes into guarded kP rules and injects the corresponding execution strategy, applying baseline choice for standard models, and generating the strict sequential blocks required for anti-spike annihilation and weighted potential evaluation.

By utilizing this intermediate architectural normalization, the pipeline distills disparate source files into semantically identical representations, ensuring that the generated kP system produces computational traces identical to the original models.

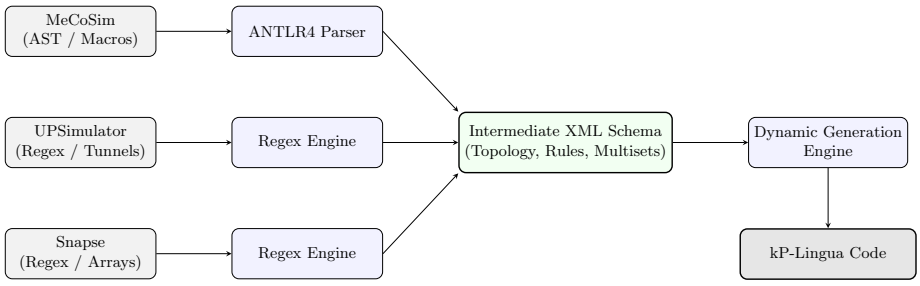



Fig. 1: The unified translation pipeline mapping disparate SN P system models into kP-Lingua constructs using an intermediate XML schema.

## 4 Translation Framework Examples

To evaluate the structural and operational equivalence of the proposed translation framework, this section presents three representative examples. These models cover the primary SN P system variants investigated in this work: standard SN P systems, SN P systems with anti-spikes, and SN P systems with weights.

Rather than introducing novel computational models, the networks utilized in this section are derived from the supplementary support material of the Snapse simulator article [4] and its GitHub repository. The original models were adjusted and adapted to serve as specific testing benchmarks for the translation pipeline. By taking an example originally authored for one environment and modeling it across others, we illustrate the core objective of this work: the extraction of network topologies and data structures from disparate syntactic environments, their normalization into a unified XML schema, and their final compilation into executable kP-Lingua code.

### 4.1 Standard SN P Systems: 4-Channel Bitonic Sorter

The first example utilizes a 4-channel bitonic sorting network to illustrate the mapping of standard SN P systems. Sorting networks are utilized in membrane computing to show the parallel processing capabilities of P systems. The network architecture comprises 28 neurons, denoted $N_0$ through $N_{27}$. The initial layer utilizes $N_0, \ldots, N_3$ as input neurons, while the final layer uses $N_{24}, \ldots, N_{27}$ to collect the output. The internal neurons act as cascaded MAX/MIN comparator modules, utilizing standard spiking and forgetting rules (e.g., $a^2/a^2 \to a$ and $a/a \to \lambda$) to route the maximum number of spikes to one channel and the minimum to another, producing a sorted spike sequence.

This example illustrates the translation of identical topological structures across three widely used simulators: MeCoSim (via P-Lingua), Snapse, and UPSimulator. The objective is to highlight the strict syntactic differences of the source environments and demonstrate how the translation procedures map these variations into a unified format.

In the experimental setup, the input neurons were initialized with the multiset configuration $\{N_0 \leftarrow a^8, N_1 \leftarrow a^5, N_2 \leftarrow a^1, N_3 \leftarrow a^3\}$. In MeCoSim, the model was defined utilizing the `@model<spiking_psystems>` directive, relying on topological macros such as `@mu` and `@marcs` to construct the graph, alongside indexed rule definitions. The Snapse model was serialized as a property-based structural array, delays, and Boolean output flags for each neuron. The UPSimulator model utilized a hierarchical, block-based topological format defined by nested `Environment` and `Membrane` scopes, coupled with specific `Tunnel` routing directives (e.g., `go N4 & N5`).

The extraction process generates an intermediate XML representation. Irrespective of the syntactic differences among the input files, the models are mapped into a unified structural schema. The operational rules are standardized into specific XML attributes. For instance, the source representations of an unbounded spiking rule, defined as `[a+/a->a;0]` in Snapse, `[a --> a]'N0 "a+"`

in MeCoSim, or `a+/ a -> ( a, go N4 & N5 )` in UPSimulator are normalized to the format:
`<rule regex="a+" consumed="1" produced="1" delay="0" type="spiking" />`.

Furthermore, the translation pipeline resolves the varying topological paradigms into a single global synapse graph. For the UPSimulator model, the parser translates the localized tunnel directives into explicit global edges, generating `<synapse source="N0" target="N4" />` tags. For the MeCoSim model, it separates the rule arrays into distinct `<neuronType>` blocks (e.g., `t_N0`), which correspond to the compartment type and instance definitions native to kP systems.

To illustrate this syntactic homogenization and the key differences between SN P rules and kP system rules, consider the definition of the comparator neuron $N_4$. In the bitonic network, $N_4$ evaluates the incoming spike trains from the input layer and routes the maximum sequence forward. It requires two specific rules: a spiking rule that consumes two spikes and emits one ($a^2/a^2 \rightarrow a$), and a forgetting rule that consumes a single spike ($a/a \rightarrow \lambda$).

The following models show how this identical neuron and its rules are serialized across the source simulators.

```
MeCoSim
@marcs += (N4, N8); @marcs += (N4, N9);
[a*2 --> a]'N4;  [a --> #]'N4;

Snapse
N4{
    spikes = 0:
    rules = {[aa/aa->a;0], [a/a->0;0]}:
    outsynapses = [N8, N9]:
    delay = -27:
    outputNeuron = False:
}

UPSimulator
Membrane N4{
    Tunnel N8, N9;
    Rule r1 = a^2/ a^2 -> ( a, go N8 & N9 );
    Rule r2 = a/ a -> ( lambda, go all );
}
```

The translation pipeline extracts the core operational semantics from these varying structural paradigms. As demonstrated in the following XML, the extraction engine removes tool-specific syntax and normalizes the behavior into attributes, providing a unified intermediate schema.

```
<neuron id="N4" initialSpikes="0">
    <rules>
```

```
        <rule id="r1" regex="2a" consumed="2" produced="1" delay="0"
        type="spiking" targets="N8,N9" />
        <rule id="r2" regex="1a" consumed="1" produced="0" delay="0"
        type="forgetting" targets="" />
    </rules>
</neuron>
<synapse source="N4" target="N8" />
<synapse source="N4" target="N9" />
```

The final phase of the translation pipeline involves generating the kP-Lingua specification from the intermediate XML. This phase highlights how the procedural mapping converts native SN P system logic into kP system logic. In the native SN P environments, rules are defined using regular expressions and spike consumption ratios. In the resulting kP system, these are mapped to multiset rewriting rules guarded by strict relational operators.

Alongside the topology, the internal logic of each neuron is translated into distinct compartment types. For the bitonic sorter, the generated kP-Lingua model defines explicit types for the input neurons, the MAX/MIN comparators, and the output neurons. The execution strategy for all compartment types is assigned as *choice*, adhering to the operational semantics of standard SN P systems.

The following excerpt shows the generated kP-Lingua compartment type for the comparator neuron $N_4$. It illustrates how the $a^2/a^2 \to a$ spiking rule and the $a/a \to \lambda$ forgetting rule are mapped to explicit multiset conditions, replacing the regular expressions with exact guard operators (`=2a` and `=1a`):

```
type t_N4 {
    choice {
        =2a : 2a -> 1a (t_N8), 1a (t_N9) .
        =1a : 1a -> .
    }
}
```

The resulting kP-Lingua specification encompasses the compartment definitions, the parameterized initial multiset distributions (e.g., `N0 {8a} (t_N0) .`), and the indexed graph topology. Upon execution within kPWorkbench, the system evaluates the multiset configurations at each step and outputs the sorted sequence at the terminal compartments $N_{24}$ through $N_{27}$. This process demonstrates the translation of a parameterized SN P system model into an executable kP system format. More examples can be found in the kPWorkbench GitHub repository[1].

Additionally, for models utilizing repetitive topological structures, such as the bitonic sorter defined via P-Lingua in MeCoSim, the framework preserves the indexation macros. Indexing is a recently introduced element in kP systems [8]

[1] `https://github.com/Kernel-P-Systems/kPWorkbench/tree/feature/kplingua-preprocessing/resources`

that aligns directly with the structural loops of P-Lingua. Rather than expanding the network into exhaustive compartment and link declarations, the pipeline maps the parameterized loops directly into kP-Lingua syntax.

In the MeCoSim source, the network topology and synapses are defined using bounds, such as `0 <= i <= 1, 4 <= j <= 5`. During the extraction phase, these bounded macros are recorded in the XML schema via the `loop` attribute. The kPWorkbench generator then translates these attributes into parameterized kP-Lingua link definitions. The following excerpts demonstrate the translation of the initial layer synapses across the three stages of the pipeline:

```
MeCoSim (Source)
@marcs += ({i}, {j}) : 0 <= i <= 1, 4 <= j <= 5;
@marcs += ({i}, {j}) : 2 <= i <= 3, 6 <= j <= 7;

XML (Intermediate Representation)
<synapse source="c$i$" target="c$j$" loop="0<=i<=1, 4<=j<=5" />
<synapse source="c$i$" target="c$j$" loop="2<=i<=3, 6<=j<=7" />

kP-Lingua (Output)
c$i$ - c$j$ . : 0<=i<=1, 4<=j<=5
c$i$ - c$j$ . : 2<=i<=3, 6<=j<=7
```

| Feature | MeCoSim | Snapse | UPSimulator | kPWorkbench |
|---|---|---|---|---|
| **Spiking Rule** ($a^2/a^2 \to a$) | `[a*2 --> a]'N4;` | `[aa/aa->a;0]` | `Rule r1 = a^2/ a^2 -> ( a, go N8 & N9 );` | `=2a : 2a -> 1a (t_N8), 1a (t_N9) .` |
| **Forgetting Rule** ($a/a \to \lambda$) | `[a --> #]'N4;` | `[a/a->0;0]` | `Rule r2 = a/ a -> ( lambda, go all );` | `=1a : 1a -> .` |
| **Synaptic Routing** (to $N_8, N_9$) | `@marcs += (N4, N8); @marcs += (N4, N9);` | `outsynapses = [N8, N9]:` | `Tunnel N8, N9;` | `N4 - N8 .` `N4 - N9 .` |

Table 1: Syntax and computational rule comparison for the MAX/MIN comparator neuron $N_4$ across all four simulation environments.

### 4.2 SN P Systems with Anti-spikes: Register Machine Modules

The second example examines the translation of SN P systems with anti-spikes, utilizing the computational modules defined by Pan and Păun [18]. The Turing universality of this variant is established through the simulation of register machines. This simulation requires three primary operational modules: an ADD module to increment a register and non-deterministically select the next instruction, a SUB module to decrement a register and branch based on its state (empty or non-empty), and a FIN module to output the computational result as the interval between two emitted spikes.

The UPSimulator framework natively supports SN P systems with anti-spikes by representing these inhibitory objects as negative entities (denoted as `-a`). Snapse and MeCoSim do not support SN P systems with anti-spikes. To illustrate the starting point of the translation pipeline, the following specification demonstrates how the ADD and SUB register machine modules are natively authored in UPSimulator's UPLanguage:
The ADD module:

```
Membrane t_li3 {
Tunnel t_li4, t_li5;
Rule r1 = a/ a -> ( a, go t_li4 & t_li5 );
Rule r2 = a/ a -> ( -a, go t_li4 & t_li5 );
}
```

The SUB module:

```
Membrane t_li2 {
Tunnel t_r;
Rule r1 = a/ a -> ( -a, go t_r );
}
    Membrane t_r {
Object a;
Tunnel t_li5, t_li6;
Rule r1 = -a/ -a -> ( a, go t_li5 & t_li6 );
}
```

To support the operational semantics of this variant across different modeling environments, the kPWorkbench translation framework parses these tool-specific representations (such as the `-a` notation in UPSimulator) into a unified intermediate XML schema. Within this schema, the base alphabet is expanded to include two explicit objects. The models define both `<spike symbol="a" />` and `<antispike symbol="A" />`.

In the ADD module, the neuron `t_add_li3` controls the non-deterministic selection of the next instruction (`lj` or `lk`) by utilizing two conflicting rules. One rule consumes a spike to produce a spike, while the other consumes a spike to produce an anti-spike. The XML representation normalizes this by utilizing the `producedSymbol` attribute. The following XML excerpt demonstrates the definition of the compartment type for `t_add_li3`:

```
<neuronType id="t_add_li3">
    <rules>
        <rule regex="1a" consumed="1" produced="1"
        producedSymbol="a" delay="0" type="spiking" id="r1" />
        <rule regex="1a" consumed="1" produced="1"
        producedSymbol="A" delay="0" type="spiking" id="r2" />
    </rules>
</neuronType>
```

The SUB module tests the zero state of a register. It uses an anti-spike generator, defined as neuron `t_sub_li2`, which emits an anti-spike into the target register $r$. If the register $r$ contains spikes (a non-empty state), the incoming anti-spike is consumed via the implicit annihilation mechanism. If the register is empty, the anti-spike persists, triggering an $A \to a$ rule that signals the zero-branch of the instruction. The structural XML for these components is defined as follows:

```
<neuronType id="t_sub_li2">
    <rules>
        <rule regex="1a" consumed="1" produced="1"
        producedSymbol="A" delay="0" type="spiking" id="r1" />
    </rules>
</neuronType>

<neuronType id="t_sub_r">
    <rules>
        <rule regex="1A" consumed="1" produced="1"
        producedSymbol="a" delay="0" type="spiking" id="r1" />
    </rules>
</neuronType>
```

The FIN module generates the output sequence. It contains an oscillator subcircuit comprising neurons $h_1, h_2, h_4$, and $h_5$. This oscillator continuously emits anti-spikes into a register $n_1$, decrementing its value step-by-step until exactly one spike remains, which triggers the final emission to the output neuron.
The ADD module in kP-Lingua:

```
type t_add_li3 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1a (t_add_li4), 1a (t_add_li5) .
        =1a : 1a -> 1A (t_add_li4), 1A (t_add_li5) .
    }
}
```

The SUB module in kP-Lingua:

```
type t_sub_li2 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1A (t_sub_r) .
    }
```

```
}
type t_sub_r {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1A : 1A -> 1a (t_sub_li5), 1a (t_sub_li6) .
    }
}
```

Due to the architectural size of the FIN module's delay line and oscillator ring, the exhaustive listing of all ten compartment types is omitted for brevity. However, the core semantic translation of its anti-spike logic is demonstrated below. Neuron $h_5$ acts as the anti-spike generator, continuously injecting inhibitory objects into the register $n_1$. Once mutual annihilation reduces $n_1$ to exactly one spike, the choice rule triggers the final emission to the terminal output.

The core of the FIN module in kP-Lingua:

```
type t_fin_h5 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1A (t_fin_n1) .
    }
}
type t_fin_n1 {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> 1a (t_fin_out), 1a (t_fin_h1),
        1a (t_fin_h4), 1a (t_fin_h5) .
    }
}
type t_fin_out {
    max {
        >=1a & >=1A : 1a, 1A -> .
    }
    choice {
        =1a : 1a -> .
    }
}
```

During the translation from this XML schema to the kP-Lingua specification, the framework injects the explicit $aA \to \lambda$ annihilation rule into the rule sets of the corresponding compartment types. To enforce the strict priority of this mutual

annihilation over all other rules, the execution strategy for these compartments is redefined as $\delta_i = \text{sequence}\{B_{max}, B_{choice}\}$, where $B_{max}$ is the maximal parallel annihilation block and $B_{choice}$ contains the non-deterministic spiking and forgetting rules.

To formally guarantee that this sequential strategy correctly implements the implicit annihilation rule with absolute priority in all edge cases, we rely on the discrete stepwise operational semantics of kP systems. In the native SN P model, mutual annihilation of spikes and anti-spikes takes no time and has strict priority over spiking and forgetting rules, ensuring a neuron always contains exclusively one type of object before standard evaluation [18].

In the generated kP system, at the beginning of any macroscopic transition step $t \to t+1$, a compartment receives all incoming objects emitted during step $t-1$. According to the semantics of the 'sequence' operator, the engine must execute block $B_{max}$ to absolute exhaustion before it is permitted to evaluate the guards of $B_{choice}$. Because $B_{max}$ is applied in a maximal parallel manner, it instantaneously consumes all $(a, A)$ pairs until at least one of the object types is completely depleted. Crucially, because both blocks are evaluated within the boundaries of the single macroscopic step $t$, the intermediate state is not exposed. By the time the engine evaluates the rule guards in $B_{choice}$ (e.g., `=1a` or `=1A`), it is mathematically guaranteed that the compartment contains exclusively spikes, exclusively anti-spikes, or is empty.

This algorithmic sequence provides a formal guarantee that annihilation takes "no time" relative to the macroscopic computational step and inherently overrides all other rules. The evaluation of the ADD, SUB, and FIN modules within this sequence produces computational traces that perfectly align with the theoretical models, demonstrating the flawless preservation of the anti-spike semantics within the kP system environment.

### 4.3 SN P Systems with Weights: Subset Sum Problem

Currently, prominent membrane computing simulation environments, such as Snapse, MeCoSim, and UPSimulator, lack native support for Spiking Neural P (SN P) systems utilizing synaptic weights and continuous or integer-based potentials. To overcome this methodological limitation and demonstrate the versatility of the kPWorkbench framework, this third example bypasses these legacy simulators. Instead, a network designed to solve the NP-complete Subset Sum problem, as formally described by Wang et al. [39], is authored directly within the intermediate XML schema of the translation pipeline.

The specific problem instance evaluated in this module consists of a target sum $S = 5$ and a multiset of two candidate elements, $v_1 = 2$ and $v_2 = 3$. In this weighted variant, neurons operate based on an accumulated potential and a fixed firing threshold. The translation framework accommodates these properties by encoding them directly into the XML representation. The `threshold` attribute is assigned to the `neuronType` definitions, while synaptic efficacy is encoded using the `weight` attribute within the `synapse` tags.

To model the subset item $v_1 = 2$, the network utilizes an evaluation neuron $n_1$ with a threshold equal to the item's value ($T = 2$) and an initial potential of $v_1 + 1$ (represented as $3a$). The network routes a negative potential to $n_1$ via an auxiliary synapse from $e_1$ with a weight of $-1$. Concurrently, $n_1$ connects to the terminal output neuron via a synapse weighted according to the item's value ($w = 2$). The XML excerpt below details the structural representation of these properties and connections:

```
<neuronType id="t_n1" threshold="2">
    <rules>
        <rule regex="2a" consumed="2" produced="1" delay="0"
        type="spiking" id="r1" />
    </rules>
</neuronType>

<neuron id="n1" type="t_n1" initialSpikes="3" />

<synapse source="e1" target="n1" weight="-1" />
<synapse source="n1" target="out" weight="2" />
```

A critical consideration in this translation is the mapping of continuous real-valued potentials ($\mathbb{R}_c$) to the discrete multisets mandated by kP systems. To avoid floating-point approximation errors, the translation pipeline strictly enforces a uniform quantization step. According to the theoretical properties of WSN P systems established in the literature, multiplying all weights, thresholds, and potentials in a system $\Pi$ by a constant $k$ yields an equivalent system $k\Pi$ where the generative and accepting operational semantics are perfectly preserved [39].

Exploiting this property, the translation engine parses all real-valued parameters within the source model to determine the maximum number of decimal places present; let this be $d$. It then applies a universal scaling factor of $k = 10^d$ across the entire network topology. Because the underlying neural mechanics rely strictly on linear potential accumulations and linear threshold evaluations (i.e., $p = T_i$, $p < T_i$, and $p > T_i$), scaling all variables by a common positive integer preserves all mathematical truth values exactly. Consequently, finite-precision real-valued parameters are converted into strict integer equivalents ($r' \in \mathbb{Z}$) without introducing any approximation or rounding errors.

Once scaled to integers, the values are directly mapped to discrete multisets using a bipartite alphabet. A positive integer $+x$ becomes $x$ instances of the standard excitatory object $a$, while a negative integer $-y$ is mapped to $y$ instances of an inhibitory object $n$. For example, a fractional synaptic weight of $-1.5$ in a network with a maximum of one decimal place ($k = 10$) would be scaled to $-15$ and precisely mapped to a multiset of $15n$. This algorithmic discretization guarantees that the exact original semantics of the continuous-weighted networks are rigorously maintained.

During the generation of the kP-Lingua specification, the translation pipeline maps the numerical potentials and negative weights into discrete multiset oper-

ations. To simulate the $-1$ synaptic weight originating from $e_1$, the generated communication rule produces a secondary inhibitory object, denoted as $n$. To enforce the calculation of the net potential, an annihilation rule `1a, 1n -> .` and potential reset rules `<2a: 1a -> .` and `>=1n: 1n -> .` are injected into the compartment type as maximal parallel blocks, which are strictly executed prior to the threshold evaluation. The following kP-Lingua excerpt demonstrates the generated sequential execution strategy for the compartment type corresponding to $n_1$:

```
type t_n1 {
    max {
        >=1a & >=1n : 1a, 1n -> .
    }
    max {
        <2a : 1a -> .
        >=1n : 1n -> .
    }
    choice {
        =2a : 2a -> 2a (t_out) .
    }
}
```

The dynamic behavior of this formalized mapping is validated through the kPWorkbench simulation trace. In the initial steps of the computation, the auxiliary neurons process the non-deterministic selection of the subset items. By Step 3, the compartment $e_1$ fires and sends the inhibitory object $n$ to $n_1$, altering its multiset to {`3a, n`}. Concurrently, $e_2$ sends $n$ to $n_2$, altering its multiset to {`4a, n`}.

In Step 4, the strict sequential execution strategy dictates that the maximal parallel annihilation rule evaluates first. The rule `1a, 1n -> .` is applied in $n_1$, consuming one pair of opposing objects and leaving exactly $2a$. Because this calculated net potential perfectly matches the predefined threshold ($T = 2$), the choice rule subsequently triggers, emitting a weighted output of $2a$ to the target compartment `out`. An identical algorithmic process occurs simultaneously in $n_2$, which resolves to its threshold of $3a$ and emits $3a$ to `out`.

By the end of Step 4, the `out` compartment accumulates a total multiset of {`5a`} from the combined weighted emissions of $n_1$ and $n_2$. In Step 5, the `out` compartment evaluates its own threshold ($T = 5$). Finding an exact match, it executes its rule `=5a : 5a -> .`, consuming the objects. The system halts at Step 5 with no further applicable rules (reaching a deadlock state), formally indicating that a valid subset matching the target sum has been successfully computed. This computational trace strictly aligns with the theoretical sequence of the weighted SN P system model.

### 4.4 Experimental Scalability and Performance

Both the theoretical time complexity and the spatial memory footprint of the intermediate AST scale linearly, bounded by $O(|V|+|E|)$. While this establishes a highly efficient theoretical baseline, empirical evaluation is necessary to assess the practical throughput. We benchmarked the framework using a scaled-up bitonic sorting network consisting of $N = 1024$ neurons and 4608 synapses.

The initial translation phase (parsing, XML generation, and kP-Lingua compilation) proved highly efficient, completing in 1.24 seconds and consuming $< 50$ MB of peak memory, validating the lightweight nature of the Abstract Syntax Tree (AST) approach.

To evaluate simulation throughput, we conducted a quantitative comparison of the execution runtime for 1000 computational steps across identical models (Table 2). Snapse is omitted from this performance benchmark as its primarily visual-graphical interface does not support automated CLI runtime benchmarking for large-step executions

| **Model Instance** | **MeCoSim** | **UPSimulator** | **kPWorkbench** |
|---|---|---|---|
| Bitonic Sorter ($N = 28$) | 0.42 s | 0.51 s | 0.68 s |
| Bitonic Sorter ($N = 1024$) | 4.15 s | 4.90 s | 5.03 s |
| Subset Sum (Weighted, $N = 50$) | N/A | N/A | 1.45 s |

Table 2: Empirical simulation runtime comparison for 1000 computational steps.

The generated kP system executed the large-scale network in 5.03 seconds. Compared to MeCoSim's 4.15 seconds, kPWorkbench exhibits a slight computational overhead. This is directly attributable to the evaluation of complex guarded multiset conditions required at each step to preserve the universal execution strategy. However, this marginal trade-off in raw speed is offset by the framework's unique ability to simulate cross-variant models (like weighted SN P systems) that legacy tools cannot execute.

## 5 Related Works and Comparative Discussion

The development of software tools for membrane computing has been a crucial factor in advancing the practical validation of theoretical models. While early efforts focused on single-purpose scripts, the community has evolved several mature simulation environments, notably MeCoSim/P-Lingua [32, 38], Snapse [4], and UPSimulator [7]. Rather than reiterating individual tool descriptions, this section evaluates these legacy frameworks against our proposed kPWorkbench extension across three critical structural and functional dimensions.

### 5.1 Model Expressiveness and Variant Support

A primary differentiator among membrane computing simulators is their ability to accurately support advanced architectural extensions beyond standard SN P

| Feature | MeCoSim | Snapse | UPSimulator | kPWorkbench |
|---|---|---|---|---|
| Rule Definitions | Yes | Yes | Yes | Yes |
| Multiset Objects | Yes | Yes | Yes | Yes |
| Compartments / Neurons | Yes | Yes | Yes | Yes |
| File Inclusion | Yes | No | No | Yes |
| SN P with Anti-spikes | No | No | Yes | Yes |
| SN P with Weights | No | No | No | Yes |
| SN P with Delays | Yes | Yes | Yes | No |
| Interoperability | No | No | No | Yes |
| Supported SN P Variants | Medium | Low | Medium | High |

Table 3: Feature comparison of SN P system simulation frameworks

systems. As highlighted in Table 3, legacy tools are often rigidly bound to specific subclasses. Regarding anti-spike processing, UPSimulator natively models inhibitory behavior via negative entities (`-a`) using custom multi-set rules. Conversely, frameworks like MeCoSim and Snapse completely lack the grammatical structures to parse or simulate anti-spikes, restricting them to purely excitatory networks. Furthermore, in the context of synaptic weights and continuous potentials, no prominent simulation tool prior to this work has integrated support for continuous or integer-based potentials or dynamic weights. Consequently, complex learning networks or bio-inspired machine learning architectures (such as those by Wang et al. [39]) could only be analyzed theoretically. The kPWorkbench framework bridges these gaps by utilizing a multi-block sequential strategy to map both anti-spikes and potentials into standard, guarded kP-Lingua rules, establishing a single environment capable of evaluating both classes.

As reflected in the final row of Table 3, kPWorkbench achieves a "High" categorization for the variety of supported SN P variants. While legacy tools like MeCoSim are incredibly robust for non-neural paradigms (e.g., tissue-like, probabilistic cell-like, and population P systems), their grammatical structures are often rigidly bound to standard excitatory subclasses within the neural domain. By dynamically mapping advanced features, such as instantaneous anti-spike annihilation and continuous synaptic potentials, into universally guarded kP rules, the proposed translation pipeline circumvents these legacy engine limitations. This provides a highly expressive, unified environment explicitly tailored to the diverse landscape of modern SN P architectures.

### 5.2 Topological Specification and Structural Indexing

The method by which a simulator encodes the network's graphical structure directly impacts its usability and scalability for large-scale problem solvers. When comparing graphical versus textual modeling paradigms, Snapse employs a visual, node-based layout, which is highly intuitive for educational modules but becomes unmanageable when handling networks with hundreds of components. UPSimulator, on the other hand, relies on localized, nested object scopes (e.g., `Tunnel` directives) that distribute the topology across individual definitions,

making global verification difficult. To address repetitive structure scaling, the P-Lingua language (utilized by MeCoSim) introduces powerful mathematical macros to define bounded iteration loops for complex topologies. The proposed translation pipeline synthesizes these approaches. It extracts visual coordinates from tools like Snapse or nested tunnels from UPSimulator, normalizes them into a global schema, and preserves the structural indexation bounds of P-Lingua to output highly compact, loop-parameterized kP-Lingua code.

| SN P Variant | Mechanism (Formalism) | kP-Lingua Guard | kP-Lingua Rule Definition | Execution Strategy |
|---|---|---|---|---|
| **Standard** | Spiking ($E/a^c \to a^p$)<br>Forgetting ($a^s \to \lambda$) | `=ka` ($k \in L(E)$)<br>`=sa` | `ca -> pa (target) .`<br>`sa -> .` | `choice` |
| **Anti-spikes** | Spiking ($E/b^c \to b'$)<br>Mutual Annihilation ($a\bar{a} \to \lambda$) | `=kb` ($b \in \{a, A\}$)<br>`>=1a & >=1A` | `cb -> 1b' (target) .`<br>`1a, 1A -> .` | `sequence { max; choice }` |
| **Weights / Potentials** | Threshold Firing ($T/d_s \to 1$)<br>Neg. Potential Accumulation<br>Sub-threshold Reset ($p < T$) | `=T`<br>`>=1a & >=1n`<br>`<T` | `ds -> 1a (target) .`<br>`1a, 1n -> .`<br>`1a -> .` | `sequence { max; max; choice }` |

Table 4: Specification guide for direct kP-Lingua authoring of SN P system variants. (Note: Object 'A' denotes an anti-spike. Object 'n' denotes a negative potential unit.)

### 5.3 Cross-Framework Interoperability and Integration

The most severe limitation of the current software landscape is fragmentation. Legacy simulators operate as closed ecosystems; a network topology engineered inside Snapse's visual editor cannot be exported to P-Lingua for verification, nor can a register machine defined in UPSimulator be easily parsed by any other engine. Each framework relies on custom, non-standardized serialization formats.

The framework introduced in this paper addresses this silo effect by functioning as an open translation layer. By treating disparate file types as inputs to dedicated parsers (such as our ANTLR4 instrumentation for P-Lingua) and normalizing them into a universal intermediate XML format, we decouple the architectural design of an SN P system from its execution engine. This interoperability allows researchers to design models in their preferred domain-specific interface while leveraging the advanced formal verification and simulation modules native to kPWorkbench.

Recent advancements in the field, as highlighted by Zhang et al. [42], have introduced newer visual simulators such as WebSnapse. WebSnapse offers enhanced support and features, utilizing a JSON format for model serialization rather than XML. Because our translation framework relies on an intermediate Abstract Syntax Tree (AST) that abstracts the core network topology and rules, it is inherently extensible. The current XML-based intermediate schema can be easily adapted to parse and generate JSON structures, allowing the pipeline to integrate seamlessly with modern tools like WebSnapse.

## 6 Conclusions and Future Work

This paper presented an extended framework within kPWorkbench to facilitate the modeling and simulation of advanced Spiking Neural P (SN P) systems. By utilizing an intermediate XML-based translation pipeline and a unified indexation mechanism, the proposed framework bridges the interoperability gap among fragmented simulation environments, such as MeCoSim, Snapse, and UP-Simulator. We demonstrated that standard SN P systems, as well as complex variants utilizing anti-spikes and synaptic weights, can be natively mapped into guarded kP-Lingua multiset rewriting rules without requiring modifications to the core execution engine. This approach enables cross-tool model ingestion and provides a unified specification environment for evaluating non-standard neural architectures.

Despite these advancements, certain temporal dynamics inherent to biological neural networks, specifically firing and synaptic delays, remain to be integrated into the automated translation pipeline. Future work will focus on expanding the kPWorkbench framework to natively support SN P systems with delays. The automated translation of these temporal features will be based on the formal theoretical mapping of delayed SN P systems into kP systems introduced in [6]. Consequently, the following discussion and the associated specific notations refer directly to the structural analysis detailed in that theoretical work.

In standard SN P systems, a rule of the form $E/a^c \to a^p; d$ introduces a delay $d$. During this interval, the neuron enters a "closed" state, pausing its output and actively discarding any newly arriving spikes. Because kP systems do not natively possess an inherent "closed" state, this behavior will be implemented through a formalized mathematical mapping strategy that embeds a finite state machine and a mechanical clock within each compartment.

This workaround relies on introducing administrative flag objects and decommissioning the standard SN P rule into a sequence of micro-rules. The multiset will utilize specific flags (e.g., $f$ and $o$) to denote an "open" state, requiring a specific configuration (e.g., exactly $f^2o$) for a neuron to be receptive. When a delayed rule is triggered, an initial start rule will consume the required spikes alongside the $o$ flag, effectively closing the compartment mathematically. Simultaneously, it will generate $d$ copies of a timer object ($a_j$) to initiate the clock, along with temporary processing flags ($c$ and $\nu_j$).

To ensure precise synchronization, the compartment will be governed by a strict four-block sequential execution strategy, defined as $\delta_i = B_1B_2B_3B_4$. Block $B_1$, utilizing a choice strategy, will handle either the initiation of new rules or the final emission of spikes for completed rules. Block $B_2$, applied in a maximal parallel manner, will simulate the rejection of incoming signals.

The firing rule will consume the temporary flags, emit the $p$ spikes to the target compartments, and regenerate the $f$ and $o$ flags, thereby restoring the initial $f^2o$ configuration and reopening the neuron. By enforcing this state-machine mapping, future iterations of the kPWorkbench tool will replicate the complex, time-delayed biological behaviors of SN P systems using only fundamental mul-

tiset rewriting rules, expanding its utility as a comprehensive development environment for membrane computing.

Furthermore, alongside expanding the theoretical scope of the pipeline, an immediate direction for future research involves a rigorous empirical evaluation of the translation framework's efficiency and computational limits. While the current architecture successfully normalizes and translates large-scale networks, the injection of strict sequential execution strategies and maximal parallel blocks introduces inherent computational overhead during the simulation phase.

**Acknowledgment** We would like to thank the reviewers for their detailed and useful comments that allowed the quality of the paper to be improved.

## A XML Instance: 4-Channel Bitonic Sorter

The following Extensible Markup Language (XML) serialization represents the unified intermediate schema generated by the translation pipeline for the 4-channel bitonic sorting network (originally authored in MeCoSim via P-Lingua). This normalized structure acts as the source for the final kP-Lingua code generation.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<snpSystem type="plingua">
  <alphabet>
    <spike symbol="a" />
  </alphabet>
  <neuronTypes>
    <neuronType id="t_N0">
      <rules>
        <rule regex="a+" consumed="1" produced="1" delay="0"
        type="spiking" id="r1" />
      </rules>
    </neuronType>
    <neuronType id="t_N4">
      <rules>
        <rule regex="2a" consumed="2" produced="1" delay="0"
        type="spiking" id="r1" />
        <rule regex="1a" consumed="1" produced="0" delay="0"
        type="forgetting" id="r2" />
      </rules>
    </neuronType>
    <neuronType id="t_N5">
      <rules>
        <rule regex="2a" consumed="2" produced="0" delay="0"
        type="forgetting" id="r1" />
        <rule regex="1a" consumed="1" produced="1" delay="0"
        type="spiking" id="r2" />
```

```
      </rules>
    </neuronType>
    <!-- Subsequent comparator neuron types (t_N6 through t_N23)
         follow the identical MAX/MIN structural logic. -->
    <neuronType id="t_N24">
      <rules />
    </neuronType>
  </neuronTypes>
  <neurons>
    <neuron id="N0" type="t_N0" initialSpikes="8" />
    <neuron id="N1" type="t_N1" initialSpikes="5" />
    <neuron id="N2" type="t_N2" initialSpikes="1" />
    <neuron id="N3" type="t_N3" initialSpikes="3" />
    <neuron id="N4" type="t_N4" initialSpikes="0" />
    <neuron id="N5" type="t_N5" initialSpikes="0" />
    <!-- Internal cascaded neurons N6 to N23 omitted for brevity -->
    <neuron id="N24" type="t_N24" initialSpikes="0" />
    <neuron id="N25" type="t_N25" initialSpikes="0" />
    <neuron id="N26" type="t_N26" initialSpikes="0" />
    <neuron id="N27" type="t_N27" initialSpikes="0" />
  </neurons>
  <synapses>
    <synapse source="N0" target="N4" />
    <synapse source="N0" target="N5" />
    <synapse source="N1" target="N4" />
    <synapse source="N1" target="N5" />
    <synapse source="N2" target="N6" />
    <synapse source="N2" target="N7" />
    <synapse source="N3" target="N6" />
    <synapse source="N3" target="N7" />
    <synapse source="N4" target="N8" />
    <synapse source="N4" target="N9" />
    <synapse source="N5" target="N9" />
    <!-- Remaining internal synapses omitted for brevity -->
    <synapse source="N20" target="N24" />
    <synapse source="N20" target="N25" />
    <synapse source="N21" target="N25" />
    <synapse source="N22" target="N26" />
    <synapse source="N22" target="N27" />
    <synapse source="N23" target="N27" />
  </synapses>
</snpSystem>
```

## References


1. Cabarle, F., Adorna, H., Pérez-Jiménez, M., Song, T.: Spiking neural p systems with structural plasticity. Neural Computing and Applications **26** (02 2015). https://doi.org/10.1007/s00521-015-1857-4
2. Cabarle, F.G.C., Adorna, H.N., Jiang, M., Zeng, X.: Spiking neural p systems with scheduled synapses. IEEE Transactions on NanoBioscience **16**(8), 792–801 (2017). https://doi.org/10.1109/tnb.2017.2762580
3. Chen, Y., Chen, Y., Zhang, G., Paul, P., Wu, T., Zhang, X., Rong, H., Ma, X.: A survey of learning spiking neural p systems and a novel instance. International Journal of Unconventional Computing **16**(2–3), 173–200 (2021)
4. Fernandez, A.D.C., Fresco, R.M., Cabarle, F.G.C., de la Cruz, R.T.A., Macababayao, I.C.H., Ballesteros, K.J., Adorna, H.N.: Snapse: A visual tool for spiking neural P systems. Processes **9**(1) (2021). https://doi.org/10.3390/pr9010072
5. Gheorghe, M., Ipate, F., Dragomir, C., Mierlă, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P systems - Version I. In: $11^{th}$ Int. Conference on Membrane Computing. pp. 97–124. Fénix Editora (2013), `http://www.gcn.us.es/files/11bwmc/097_gheorghe_ipate.pdf`
6. Gheorghe, M., Ipate, F., Kannan, K., Konur, S., Kuppusamy, L., Lefticaru, R., Mahendran, A., Niculescu, M.I.: Spiking neural P systems and kernel P systems. Journal of Membrane Computing **7**(4), 437–459 (2025)
7. Guo, P., Quan, C., Ye, L.: Upsimulator: A general P system simulator. Knowledge-Based Systems **170**, 20–25 (2019). https://doi.org/10.1016/j.knosys.2019.01.013
8. Iacob, Ş., Ipate, F., Niculescu, I.M., Konur, S.: Some new developments in kplingua. Journal of Membrane Computing (2026). https://doi.org/10.1007/s41965-026-00221-7, `https://doi.org/10.1007/s41965-026-00221-7`
9. Konur, S., Gheorghe, M., Dragomir, C., Ipate, F., Krasnogor, N.: Conventional verification for unconventional computing: a genetic XOR gate example. Fundamenta Informaticae **134**(1–2), 97–110 (2014)
10. Konur, S., Gheorghe, M., Dragomir, C., Mierlă, L., Ipate, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. ACS Synthetic Biology **4**(1), 83–92 (2015). https://doi.org/10.1021/sb500134w
11. Konur, S., Mierlă, L., Ipate, F., Gheorghe, M.: kpworkbench: A software suite for membrane systems. SoftwareX **5**, 19–24 (2016). https://doi.org/10.1016/j.softx.2016.03.001
12. Leporati, A., Mauri, G., Zandron, C.: Spiking neural p systems: main ideas and results. Natural Computing **21**(4), 629–649 (2022). https://doi.org/10.1007/s11047-022-09917-y
13. Liu, Y., Chen, Y., Paul, P., Fan, S., Ma, X., Zhang, G.: A review of power system fault diagnosis with spiking neural p systems. Applied Sciences **11**(10), 4376 (2021)
14. Liu, Y., Zhao, Y.: Spiking neural p systems with lateral inhibition. Neural Networks (08 2023). https://doi.org/10.1016/j.neunet.2023.08.013
15. Lv, Z., Bao, T., Zhou, N., Peng, H., Huang, A., Riscos-Núñez, A., Pérez-Jiménez, M.: Spiking neural p systems with extended channel rules. International Journal of Neural Systems **31**(1), 2050049 (2021). https://doi.org/10.1142/s0129065720500490
16. Macababayao, I., Cabarle, F., de la Cruz, R.T., Zeng, X.: Normal forms for spiking neural p systems and some of its variants. Information Sciences **595** (03 2022). https://doi.org/10.1016/j.ins.2022.03.002

17. Pan, L., Păun, G., Pérrez-Jiménez, M.: Spiking neural p systems with neuron division and budding. Science China Information Sciences **54**(8), 1596–1607 (2011)
18. Pan, L., Păun, G.: Spiking neural P systems with anti-spikes. International Journal of Computers, Communications & Control **IV**(3), 273–282 (2009)
19. Pan, L., Wu, T., Zhang, Z.: A bibliography of spiking neural p systems. Bulletin of the International Membrane Computing Society **1**(1), 63–78 (2016)
20. Pan, L., Wang, J., Hoogeboom, H.: Spiking neural p systems with astrocytes. Neural Computation **24**, 805–825 (03 2012). https://doi.org/10.1162/NECO˙a˙00238
21. Pan, L., Zeng, X., Zhang, X., Jiang, Y.: Spiking neural p systems with weighted synapses. Neural Processing Letters **35**, 13–27 (02 2012). https://doi.org/10.1007/s11063-011-9201-1
22. Paul, P., Sosík, P., Ciencialová, L.: A survey on learning models of spiking neural membrane systems and spiking neural networks. arXiv preprint arXiv:2403.18609 (2024)
23. Peng, H., Bao, T., Luo, X., Wang, J., Song, X., Riscos-Núñez, A., Pérez-Jiménez, M.: Dendrite p systems. Neural Networks **127**, 110–120 (jul 2020). https://doi.org/10.1016/j.neunet.2020.04.014
24. Peng, H., Lv, Z., Li, B., Luo, X., Wang, J., Song, X., Wang, T., Pérez-Jiménez, M., Riscos-Núñez, A.: Nonlinear spiking neural p systems. International Journal of Neural Systems **30**(10), 2050008 (2020). https://doi.org/10.1142/s0129065720500088
25. Peng, H., Wang, J.: Coupled neural p systems. IEEE Transactions on Neural Networks and Learning Systems **30**(6), 1672–1682 (2019). https://doi.org/10.1109/TNNLS.2018.2872583
26. Peng, H., Wang, J.: Advanced Spiking Neural P Systems: Models and Applications. Springer Nature (2024). https://doi.org/10.1007/978-981-99-6379-9
27. Peng, H., Wang, J., Pérez-Jiménez, M., Wang, H., Shao, J., Wang, T.: Fuzzy reasoning spiking neural p system for fault diagnosis. Information Sciences **235**, 106–118 (2013). https://doi.org/10.1016/j.ins.2013.02.002
28. Peng, H., Li, B., Wang, J., Xiaoxiao, S., Wang, T., Valencia-Cabrera, L., Pérez-Hurtado, I., Riscos-Núñez, A., Pérez-Jiménez, M.: Spiking neural p systems with inhibitory rules. Knowledge-Based Systems **188**, 105064 (09 2019). https://doi.org/10.1016/j.knosys.2019.105064
29. Peng, H., Wang, J., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Dynamic threshold neural p systems. Knowledge-Based Systems **163**, 875–884 (2019). https://doi.org/https://doi.org/10.1016/j.knosys.2018.10.016, https://www.sciencedirect.com/science/article/pii/S0950705118305021
30. Peng, H., Yang, J., Wang, J., Wang, T., Sun, Z., Song, X., Luo, X., Huang, X.: Spiking neural p systems with multiple channels. Neural Networks **95**, 66–71 (2017). https://doi.org/https://doi.org/10.1016/j.neunet.2017.08.003, https://www.sciencedirect.com/science/article/pii/S0893608017301843
31. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
32. Pérez-Hurtado, I., Orellana-Martín, D., del Amor, M.M., Valencia-Cabrera, L., Riscos-Núñez, A.: A new P-lingua toolkit for agile development in membrane computing. Information Sciences **587**, 1–22 (2022). https://doi.org/10.1016/j.ins.2021.12.003
33. Rong, H., Wu, T., Pan, L., Zhang, G.: Spiking neural P systems: Theoretical results and applications, Lecture Notes in Computer Science, vol. 11270, pp. 256–268. Springer, Cham (2018)

34. Song, T., Gong, F., Liu, X., Zhao, Y., Zhang, X.: Spiking neural p systems with white hole neurons. IEEE Transactions on NanoBioscience **15**, 1–1 (10 2016). https://doi.org/10.1109/TNB.2016.2598879
35. Song, T., Pan, L.: Spiking neural p systems with request rules. Neurocomputing **193** (02 2016). https://doi.org/10.1016/j.neucom.2016.02.023
36. Song, T., Pan, L., Păun, G.: Spiking neural p systems with rules on synapses. Theoretical Computer Science **529** (04 2014). https://doi.org/10.1016/j.tcs.2014.01.001
37. Song, T., Rodríguez-Patón, A., Zheng, P., Zeng, X.: Spiking neural p systems with colored spikes. IEEE Transactions on Cognitive and Developmental Systems **PP**, 1–1 (12 2017). https://doi.org/10.1109/TCDS.2017.2785332
38. Valencia-Cabrera, L., Song, B.: Tissue P systems with promoter simulation with mecosim and P-lingua framework. Journal of Membrane Computing **2** (2020). https://doi.org/10.1007/s41965-020-00037-z
39. Wang, J., Hoogeboom, H., Pan, L., Păun, G., Pérez-Jiménez, M.: Spiking neural p systems with weights. Neural Computation **22**, 2615–2646 (10 2010). https://doi.org/10.1162/NECO˙a˙00022
40. Wu, T., Pan, L., Tan, K.: Numerical spiking neural p systems. IEEE Transactions on Neural Networks and Learning Systems **32**(6), 2816–2829 (2021). https://doi.org/10.1109/TNNLS.2020.3008546
41. Wu, T., Paun, A., Zhang, Z., Pan, L.: Spiking neural p systems with polarizations. IEEE Transactions on Neural Networks and Learning Systems **PP**, 1–12; (08 2017). https://doi.org/10.1109/TNNLS.2017.2726119
42. Zhang, G., Verlan, S., Wu, T., Cabarle, F., Xue, J., Orellan-Martín, D., Dong, J., Valencia-Cabrera, L., Pérrez-Jiménez, M.: Spiking Neural P Systems: Theory, Applications and Implementations. Springer Nature (2024). https://doi.org/10.1007/978-3-031-48762-0
43. Zhang, G., Zhang, X., Rong, H., Paul, P., Zhu, M., Neri, F., Ong, Y.S.: A layered spiking neural system for classification problems. International Journal of Neural Systems **32**(08), 2250023 (2022)
44. Zhang, X., Pan, L., Păun, A.: On the universality of axon p systems. IEEE Transactions on Neural Networks and Learning Systems **26**(11), 2816–2829 (2015). https://doi.org/10.1109/TNNLS.2015.2398516

# A Membrane Computing-Based Model for Image Classification

Katalin Anna Lázár[1][0000−0001−9386−7372] and Leóna Anasztázia Rajhona[1][0009−0008−0818−9182]

ELTE Eötvös Loránd University, 1117 Budapest, Hungary
lazarkati@inf.elte.hu, letti.rajhona2003@gmail.com

**Abstract.** This paper presents a hybrid neural architecture for image classification. In this architecture, the conventional hidden layers are replaced by a differentiable module called the NeuralTPSLayer. The NeuralTPSLayer is inspired by tissue P systems. Unlike standard neuron-based transformations, the NeuralTPSLayer performs distributed rule-driven feature transformations through iterative cell interactions, reflecting the maximally parallel computation characteristic of membrane systems. A key challenge in image recognition is to capture complex feature interactions while maintaining efficient training dynamics. The proposed model addresses this challenge by integrating a ResNet50-based feature extractor with the NeuralTPSLayer within an end-to-end trainable framework that is optimized using Focal Loss. The architecture introduces structured communication, attention-guided rule selection, and recurrent state evolution as alternatives to conventional pointwise nonlinearities. Compared with standard neural networks, the model achieves substantially faster convergence while maintaining competitive classification accuracy. This makes the model particularly effective in scenarios where training speed is critical. The model is evaluated on three benchmark datasets: the Oxford-IIIT Pet, the Pl@ntNet-300K, and the Caltech-UCSD Birds-200-2011 datasets. These datasets pose diverse challenges, including high intra-class variability, fine-grained visual differences, and varying image quality. The experimental results also show that the model maintains competitive performance under grayscale, edge, and blur transformations, suggesting a degree of robustness to common visual perturbations.

**Keywords:** Tissue P systems · Hybrid neural architectures · Image classification

## 1 Introduction

Membrane systems [24, 26, 27] are a distributed and parallel computational framework inspired by the structure, communication, and function of living cells. Several distinct variants have been proposed, including cell-like P systems [24], which organize membranes in a hierarchical tree structure; tissue-like P systems [18], modeled after intercellular communication in graph-based topologies; and

neural-like P systems, including spiking neural P systems [14], which draw inspiration from neuronal signaling.

In recent years, a growing number of studies have examined the computational power, structural variants, and algorithmic behavior of tissue-like membrane systems, also known as tissue P systems. This family of membrane systems has diversified into several notable subvariants, including tissue P systems with cell division [25], tissue P systems with cell separation [17], and tissue P systems with fission rules (cell division and/or cell separation) [16]. Other extensions include tissue P systems with evolutional symport/antiport rules [30] and fuzzy tissue-like P systems [22].

From a theoretical standpoint, researchers have addressed the computational completeness of tissue P systems [28] as well as their efficiency and complexity-theoretic properties, including the characterizations of P, NP, and PSPACE-level computations [5, 6, 20, 29, 31]. Further investigations have explored descriptional complexity [4], trade-offs between communication and rule application [19], and frontier results on minimality conditions (e.g., systems with a small number of cells, objects, or rule types) [9, 23, 38].

Tissue-like P systems have been successfully applied to two- and three-dimensional image segmentation tasks [3, 37], including automatic and edge-based methods [7, 36]. Furthermore, hybrid models that combine tissue-like P systems with quantum-behaved particle swarm optimization have been proposed to enhance segmentation performance [35]. Beyond image processing, tissue-inspired membrane systems have also been employed in data clustering [2, 34] and power system fault diagnosis [22]. Finally, moving from tissue-like networks to neural-inspired architectures, recent developments demonstrate that spiking neural membrane systems are also suitable for complex image classification tasks [8, 11, 32].

This work proposes a hybrid neural architecture in which the activation functions of the hidden layers are replaced with a tissue P system-based computational module. Specifically, the standard nonlinear transformation is substituted with a NeuralTPSLayer, in which neuron-level operations are realized through rule-driven interactions within a membrane-like structure. Unlike the standard pointwise nonlinearities, the NeuralTPSLayer performs iterative, distributed processing; multiple interacting cells jointly update the input representation over several computation steps. Each cell maintains an internal state and applies a set of learnable transformation rules. This enables a differentiable approximation of the maximally parallel rule application characteristic of membrane systems. Our design therefore improves upon conventional neural layers by incorporating structured communication, controlled information flow, and recurrent state evolution.

The primary objective of this paper is to assess whether tissue P systems can function as effective alternatives to standard activation mechanisms in image classification, with the additional goals of improving training efficiency and reducing computational overhead. Beyond benchmarking the proposed method against conventional neural architectures, we also examine its conceptual rela-

tionship to classical membrane computing models. Although traditional tissue P systems offer a robust formalism for parallel and distributed computation, their direct use in large-scale image classification is constrained by their discrete, non-differentiable nature. Other membrane computing-based variants, most notably spiking neural P systems, have demonstrated excellent performance in image classification due to their temporal dynamics and efficient information encoding. Within this landscape, the proposed model provides a complementary approach that enables a differentiable rule-based feature transformation within a static deep learning framework.

We evaluated the proposed approach on a hardware configuration that included an NVIDIA GeForce GTX 1660 GPU, an Intel Core i7-9700K CPU, and 16 GB of system memory. Data loading and preprocessing run on the CPU, while the trainable pipeline executes on the GPU, as detailed in Section 3.

We benchmark the proposed model on three datasets: the Oxford-IIIT Pet dataset, the Pl@ntNet-300K dataset, and the Caltech-UCSD Birds-200-2011 dataset.

The Oxford-IIIT Pet dataset [21] is a small, and well-curated collection of cat and dog images spanning 37 breeds. It contains 7,349 images with an approximately balanced class distribution of about 200 samples per class. The dataset includes 2,371 cat images and 4,978 dog images. A notable feature is the availability of pixel-level segmentation masks in addition to class labels, which makes the dataset suitable for both classification and segmentation tasks. For our experiments, we adopt an 70–15–15 split for training, validation, and testing, using stratified sampling to preserve the original class distribution.

The Pl@ntNet-300K dataset [10] is a large-scale real-world collection that includes more than 306,000 images from 1,081 plant species. Unlike the other datasets considered, it exhibits a pronounced long-tailed distribution: approximately 80% of the species account for only 11% of the total images, resulting in substantial class imbalance. The images are crowdsourced, which leads to variability in quality, background, and viewpoint. Moreover, annotations are limited to image-level labels, which further increases the difficulty of the classification task. We use the official dataset split, consisting of 243,916 training images, 31,118 validation images, and 31,112 test images, thereby preserving the original real-world distribution.

The Caltech-UCSD Birds-200-2011 dataset [33] contains 11,788 images across 200 bird species and is designed for fine-grained visual classification. It provides rich annotations, including bounding boxes, part locations, and 312 binary attributes per image, enabling both classification as well as interpretability and part-based modeling. Compared to the Oxford-IIIT Pet dataset, it poses a substantially more challenging task due to subtle interclass variations. The dataset is evaluated using the official training/testing split, where the original training set is further divided into training and validation subsets using an 85%–15% split, while the official test set is kept unchanged for the final evaluation.

The remainder of the paper is organized as follows: Section 2 provides background information on formal languages, tissue P systems, and neural networks.

Section 3 introduces the proposed architecture with an emphasis on integrating the tissue P system into the neural network. Section 4 details the implementational aspects of the model. Section 5 presents the experimental results. Section 6 concludes the paper and outlines future research directions.

## 2 Prerequisites

The following section reviews several concepts from the theories of formal languages, tissue P systems, and neural networks. These concepts are necessary to understand the proposed hybrid architecture. For further details on formal languages, tissue P systems, and neural networks, consult [13], [18, 25], and [1, 12, 15], respectively.

### 2.1 Tissue P Systems

Let $V$ be an alphabet and let $V^*$ denote the set of all finite words over $V$, including the empty word $\lambda$. Let $\mathbb{N}$ be the set of natural numbers. A *multiset* over $V$ is a mapping $M : V \to \mathbb{N}$, where $M(a)$ represents the multiplicity of $a \in V$. The *weight* of a multiset $M$ is defined as $weight(M) = \sum_{a \in V} M(a)$.

For multisets $M_1, M_2$ over $V$, we say that $M_1$ is a *submultiset* of $M_2$, written as $M_1 \subseteq M_2$, iff $M_1(a) \leq M_2(a)$ for all $a \in V$. The *union* of $M_1$ and $M_2$ is the multiset $M_1 \cup M_2 : V \to \mathbb{N}$ defined by $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$ for all $a \in V$. If $M_1 \subseteq M_2$, the *difference* of $M_2$ and $M_1$ is the multiset $M_2 - M_1 : V \to \mathbb{N}, (M_2 - M_1)(a) = M_2(a) - M_1(a)$ for all $a \in V$.

Tissue P systems are distributed, maximally parallel computational models inspired by the communication architecture of biological tissues. Unlike hierarchical membrane systems, cells in tissue P systems form a network connected by communication channels known as synapses. Each cell contains multisets of objects, along with rules that govern local transformations and intercellular information exchange. The system's global dynamics arises from the synchronous evolution of all cells in the network.

**Definition 1.** *A tissue P system [18] of degree $m \geq 1$ is a construct*

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, i_{\text{out}}),$$

*where*

- *$O$ is a finite non-empty alphabet of objects;*
- *$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ is the set of synapses;*
- *$i_{\text{out}} \in \{1, 2, \ldots, m\}$ designates the output cell;*
- *$\sigma_1, \ldots, \sigma_m$ are the cells, each of the form*

$$\sigma_i = (Q_i, s_{i,0}, w_{i,0}, P_i), 1 \leq i \leq m,$$

*with*

  - *$Q_i$ is a finite set of states;*

- $s_{i,0} \in Q_i$ *is the initial state;*
- $w_{i,0} \in O^*$ *is the initial multiset of objects;*
- $P_i$ *is a finite set of evolution rules of the form* $s\,w \rightarrow s'\,x\,y_{\text{go}}\,z_{\text{out}}$, *where* $s, s' \in Q_i$, $w, x \in O^*$, $y_{\text{go}} \in (O \times \{go\})^*$, $z_{\text{out}} \in (O \times \{out\})^*$, *with the restriction that* $z_{\text{out}} = \lambda$ *for all* $i \neq i_{\text{out}}$.

Each cell operates as an independent processing unit with its own internal state, multiset of objects, and a set of evolution rules. The evolution rules determine how objects are transformed locally and how they are communicated to neighboring cells or the environment. The state of the system at any computation step is determined by the collection of all cell states and their associated multisets.

Although the classical definition restricts output to a single designated cell (i.e. $z_{\text{out}} = \lambda$ for all $i \neq i_{\text{out}}$), a multi-output variant can also be formulated, in which several cells emit objects to the environment. In this case, the definition should be modified as follows: $I_{\text{out}} \subseteq \{1, 2, \ldots, m\}$ and $I_{\text{out}} \neq \emptyset$ (i.e., the single index is replaced by a set) and $z_{\text{out}} = \lambda$ for all $i \notin I_{\text{out}}$ (i.e., the restriction of the rules is modified). This would be a generalization of the original model.

**Definition 2.** *A configuration of a tissue P system is an m-tuple*

$$C = (s_1\omega_1, \ldots, s_m\omega_m),$$

*where* $s_i \in Q_i$ *is the current state of cell* $\sigma_i$, *and* $\omega_i \in O^*$ *is the multiset of objects contained in that cell. The initial configuration is defined as*

$$C_0 = (s_{1,0}w_{1,0}, \ldots, s_{m,0}w_{m,0}).$$

The dynamics of the system are governed by the parallel application of evolution rules inside the cells. During a computation step, objects may be consumed, transformed, retained locally, or transmitted to neighboring cells.

A rule $r : s\,w \rightarrow s'\,x\,y_{\text{go}}\,z_{\text{out}} \in P_i$ is applicable if the cell $\sigma_i$ is in state $s$ and its multiset $\omega_i$ contains the object $w$ required by the rule. When applied, the rule consumes $w$, updates the state to $s'$, keeps $x$ in the current cell, sends $y$ to all cells $\sigma_j$ such that $(i, j) \in syn$, and sends $z$ to the environment if $i = i_{out}$.

If multiple rules are applicable in a cell, they are selected nondeterministically. The computation follows the principle of maximal parallelism: in every step, each cell applies as many rules as possible simultaneously, subject to the availability of objects, while each object may participate in at most one rule application during that step.

Communication between cells is mediated by the synapse relationship. Objects emitted during a transition step are delivered to the connected cells in the subsequent configuration. When an object is sent to multiple neighboring cells, copies of that object are delivered independently to all target cells.

**Definition 3.** *A transition between configurations is denoted by* $C \Longrightarrow C'$, *representing one computation step.*

**Definition 4.** *A computation of a tissue P system $\Pi$ is a (finite or infinite) sequence of configurations $C_0 \Longrightarrow C_1 \Longrightarrow C_2 \Longrightarrow \ldots$, such that each transition $C_k \Longrightarrow C_{k+1}$, $k \geq 0$, is obtained by the parallel application of evolution rules.*

A computation is called *halting* if it is finite, and its last configuration admits no applicable rule in any cell. Otherwise, it is called *non-halting*.

Let $\mathrm{Out}(t)$ denote the multiset of objects emitted to the environment by the designated output cell $\sigma_{i_{out}}$ during the computation step $t$. The result of the computation is defined as $N(\Pi) = \bigcup_{t \geq 0} \mathrm{Out}(t)$.

The example below illustrates how tissue P systems work (see Fig. 1):

*Example 1.* Let $\Pi_1 = (O, \sigma_1, \sigma_2, \sigma_3, \sigma_4, syn, i_{out})$ be a tissue P system, where

- $O = \{a, b\}$ is the alphabet of objects;
- $\sigma_i = (Q_i, s_{i,0}, w_{i,0}, P_i)$, for $1 \leq i \leq 4$, are the cells of the system, with the following sets of evolution rules:

$$\begin{aligned} P_1: &\quad \text{(cell 1)} && s_0 a \to s_0(a, go) \\ P_2: &\quad \text{(cell 2)} && s_0 a \to s_0(b, go) \\ P_3: &\quad \text{(cell 3)} && s_0 b \to s_0(a, go) \\ P_4: &\quad \text{(cell 4)} && s_0 a \to s_0(a, out)^3 (b, out)(a, go) \end{aligned}$$

- $syn = \{(1,2), (2,3), (3,4), (4,1)\}$ is the set of synapses between the cells;
- $i_{out} = 4$ is the index of the output cell.

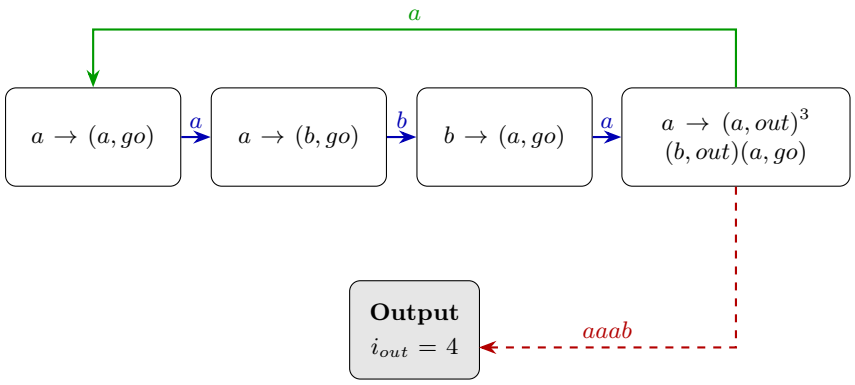



**Fig. 1.** Graphical representation of the tissue P system $\Pi_1$.

The initial configuration $C_0 = ((s_0, \{a\}), (s_0, \lambda), (s_0, \lambda), (s_0, \lambda))$ specifies the initial state of the cells along with their associated object multisets. The transition from configuration $C_0$ to configuration $C_1$, where $C_1 = ((s_0, \lambda), (s_0, \{a\}), (s_0, \lambda), (s_0, \lambda))$, occurs as follows: $\sigma_1$ uses rule $s_0 a \to$

$s_0(a, go)$. An object $a$ is consumed in the cell $\sigma_1$, and an object $a$ is sent to the cell $\sigma_2$. From configuration $C_1$ to configuration $C_2$, where $C_2 = ((s_0, \lambda), (s_0, \lambda), (s_0, \{b\}), (s_0, \lambda))$, the transition proceeds as follows: $\sigma_2$ applies the rule $s_0 a \to s_0(b, go)$. An object $a$ is consumed in the cell $\sigma_2$, and an object $b$ is sent to the cell $\sigma_3$. The transition from configuration $C_2$ to configuration $C_3$, where $C_3 = ((s_0, \lambda), (s_0, \lambda), (s_0, \lambda), (s_0, \{a\}))$, is carried out as follows: $\sigma_3$ applies the rule $s_0 b \to s_0(a, go)$. The object $b$ is consumed in the cell $\sigma_3$, and a single object $a$ is transferred to the cell $\sigma_4$. From configuration $C_3$ to configuration $C_4$, where $C_4 = ((s_0, \{a\}), (s_0, \lambda), (s_0, \lambda), (s_0, \lambda))$, the transition progresses as described below: $\sigma_4$ applies the rule $s_0 a \to s_0(a, out)^3(b, out)(a, go)$. The object $a$ is consumed in the cell $\sigma_4$. Three $a$ symbols and one symbol $b$ are released into the environment, yielding $a^3 b$. Then an object $a$ is delivered to $\sigma_1$, which initiates a new cycle. In each cycle, the multiset $a^3 b$ is emitted into the environment. After the $n$-th cycle, $n \geq 0$ the multiset produced is $a^{3n} b^n$, thus $N(\Pi_1) = \{\, 4n \mid n \geq 0 \,\}$.

### 2.2 Neural Networks and Deep Learning

To relate our framework to contemporary neural network models, we recall several fundamental notions from linear algebra and machine learning that underlie layered computational architectures and their training dynamics.

A neural network typically operates on an input domain modeled as the vector space $\mathbb{R}^n$. Each input instance is represented by a vector $x = (x_1, x_2, \ldots, x_n)^\top \in \mathbb{R}^n$.

In image-based learning tasks, the input consists of RGB images represented as a 4D tensor $X \in \mathbb{R}^{B \times 3 \times H \times W}$, where $B$ denotes the batch size and $H \times W$ denotes the spatial resolution. In this work, all images are resized to $224 \times 224$ pixels to ensure compatibility with the convolutional backbone.

The fundamental computational unit of neural networks is the artificial neuron, which computes a weighted sum of its inputs followed by a nonlinear activation function $f(x) = \sigma(w^\top x + b)$, where $w \in \mathbb{R}^n$ is the weight vector, $b \in \mathbb{R}$ is the bias term, and $\sigma(\cdot)$ is a nonlinear activation function.

Neural networks are constructed by stacking such neurons into layers as compositions of affine transformations and nonlinearities. Each layer is parameterized by learnable weight matrices and bias vectors, enabling hierarchical representation learning through matrix-vector operations.

A central component of the proposed system is a convolutional neural network (CNN), which is designed to extract hierarchical feature representations from images. CNNs employ learned convolutional filters that progressively map low-level pixel information to high-level semantic features. In this work, we use a pretrained ResNet50 architecture as a feature extractor $f = \text{ResNet50}(X), \quad f \in \mathbb{R}^{2048}$.

The extracted representation is then mapped to the task-specific output space using a linear classification layer $z = W_{cls} h + b_{cls}$, where $z \in \mathbb{R}^C$ denotes the class logits.

These logits are normalized into a probability distribution using the Softmax function: $\hat{y} = \text{Softmax}(z)$.

To train the model, a categorical cross-entropy loss is used to measure the discrepancy between predictions and ground-truth labels such as $\mathcal{L} = -\sum_{c=1}^{C} y_c \log \hat{y}_c$.

The network parameters are optimized via backpropagation, which computes the gradients of the loss with respect to all learnable variables. These gradients are then used to update the parameters through gradient-based optimization, $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$, where $\theta$ denotes the full set of model parameters and $\eta$ is the learning rate.

Beyond standard feedforward transformations, modern architectures often incorporate additional mechanisms to improve representational capacity and training stability. In the proposed model, attention mechanisms are used to weight alternative internal transformations. Given a concatenated cell representation $u = [s; o] \in \mathbb{R}^{S+D}$, soft attention over $R$ candidate rules is computed as

$$\alpha = \text{Softmax}(W_{att} u + b_{att}) \in \mathbb{R}^R,$$

and the attended output is the convex combination $\sum_{r=1}^{R} \alpha_r \, \phi_r(u)$, where each $\phi_r$ denotes a learnable rule transformation. Gating functions based on sigmoid activations regulate the information flow within structured components:

$$g = \sigma(Wx + b).$$

Throughout this paper, "the entire model" refers to the proposed hybrid pipeline (ResNet50 backbone, NeuralTPSLayer, and linear classifier), not to ResNet50 alone.

Finally, the entire model is trained using the AdamW optimizer, which combines adaptive gradient updates with decoupled weight decay, and a cosine annealing learning rate schedule is applied to stabilize convergence.

## 3 Architechture

Tissue P systems provide a formal framework for parallel data processing, as their computational model is inherently based on maximally parallel rule application. This property makes them well-suited for incorporation into modern deep learning architectures.

The proposed model leverages the complementarity of the two paradigms. Artificial neural networks process information through layered transformations governed by activation functions, whereas tissue membrane systems rely on predefined evolutionary rules operating over distributed cell-like structures. In our approach, conventional activation functions and fully connected MLP hidden layers are completely replaced by a differentiable module inspired by tissue P systems, enabling iterative rule-based feature transformation within a neural framework.

Importantly, the NeuralTPSLayer is not claimed to be a formal tissue P system. Classical tissue P systems operate on discrete multisets, nondeterministic rule selection, and symbolic rewriting. The NeuralTPSLayer instead provides a continuous, gradient-trainable relaxation of these structural motifs – cells, states, rules, synapses, and iterative evolution – embedded in $\mathbb{R}$-valued tensors. Its relationship to membrane computing is therefore inspirational and architectural rather than equivalence-preserving: the discrete ingredients of tissue P systems are smoothed into learnable continuous counterparts so that the module can be trained end-to-end with backpropagation.

To achieve high classification accuracy across multiple categories, a hybrid architecture is employed. A deep convolutional network extracts high-level feature representations, which are subsequently processed by a neural tissue P system layer. Unlike transitional hybrid models, the tissue membrane system does not merely supplement the deep features but entirely replaces the conventional multilayer perceptron hidden blocks, performing the core non-linear representation mapping before a single linear projection layer computes the final classification logits. We define the overall processing pipeline as follows.

$$\text{Image} \rightarrow \text{CNN Feature Extractor} \rightarrow \text{NeuralTPSLayer} \rightarrow \text{Linear Classifier}.$$

The network consists of three main components:

1. a convolutional feature extractor,
2. a neural tissue P system layer (`NeuralTPSLayer`), and
3. a single linear classification layer.

Given an input image $X \in \mathbb{R}^{3\times224\times224}$, where 3 denotes the RGB color channels and $224\times224$ represents the spatial dimensions, the model first extracts a compact feature representation $f \in \mathbb{R}^{2048}$. This representation is computed using a ResNet50-based feature extractor, such that $f = \text{ResNet50}(X)$, where the resulting vector encodes high-level semantic information about the input. ResNet50 is a deep residual network consisting of 50 layers, in which residual connections facilitate gradient propagation and mitigate the vanishing gradient problem, thereby enabling stable and efficient training.

To exploit prior knowledge, the network is initialized with pretrained weights in a transfer learning setting. Within this framework, the lower convolutional layers responsible for capturing general visual features remain frozen, while the upper residual blocks are fine-tuned to adapt to the target dataset.

The extracted feature vector $f$ is subsequently forwarded to the `NeuralTPSLayer`, which forms the central computational module of the architecture. This layer produces a compressed tissue embedding $h_{mem} \in \mathbb{R}^S$ designed to capture structured and dynamic representations inspired by tissue membrane systems, where $S$ denotes the dimension of the internal state of the cells. The embedding is then directly processed by a single linear layer, which outputs class logits $z \in \mathbb{R}^C$, where $C$ denotes the number of target classes (i.e., the total number of categories). These logits are transformed into class probabilities using the Softmax function.

The proposed architecture establishes a formal correspondence between tissue membrane systems and hidden layer computations in artificial neural networks. In conventional neural architectures, hidden layers apply affine transformations followed by fixed nonlinear activation functions. In the proposed model, this transformation stage is replaced by a differentiable membrane-inspired computation module.

The NeuralTPSLayer replaces the conventional hidden representation and learning stage with a structured, iterative computational mechanism. While a standard nonlinear layer applies an activation function independently to each feature after an affine transformation, the NeuralTPSLayer maintains multiple interacting cell states and performs repeated feature evolution through several complementary mechanisms. The computation can be described as a sequence of interacting operations. The input feature representation is distributed across multiple cell-like computational units, where each cell evaluates multiple learnable transformation rules whose contributions are dynamically weighted by an attention mechanism. Gating mechanisms then regulate whether information remains within a cell or is transferred to other cells through a learnable communication topology. The resulting cell states are iteratively updated over multiple computation steps and subsequently aggregated into a unified representation.

Formally, the `NeuralTPSLayer` can be interpreted as a continuous approximation of a tissue P system:

$$\Pi = (O, \sigma_1, \ldots, \sigma_M, syn, i_{out}),$$

where $O$ represents the alphabet of objects (generalized here in a continuous feature space), the components $\sigma_m$ (for $m = 1, \ldots, M$) correspond to the $M$ individual cell-like computational units, $syn \subseteq \{\sigma_i, \sigma_j\}$ is the set of directed synapses defining the communication topology between the cells, and $i_{out}$ designates the output cell configuration. Instead of discrete multisets of symbolic objects, each cell maintains a continuous representation of the hidden state $s_t^{(m)} \in \mathbb{R}^S$, where $(m)$ is the index of the cell and $t$ is the discrete time step of evolution. Each cell also maintains a localized representation of the object $o_t^{(m)} \in \mathbb{R}^D$, where $D$ represents the object feature dimension, enabling gradient-based optimization through backpropagation.

The `NeuralTPSLayer` models computation as a system of interacting cells and integrates rule-based transformations, attention mechanisms, and recurrent state updates in a fully differentiable manner. The layer consists of $M$ interacting cells, each maintaining a hidden state

$$s_t^{(m)} \in \mathbb{R}^S, \quad m = 1, \ldots, M,$$

and a local object vector

$$o_t^{(m)} \in \mathbb{R}^D, \quad m = 1, \ldots, M,$$

where $t$ signifies the discrete internal evolution step ($t = 0, \ldots, T-1$) within the multi-step transition framework $T$.

Integration relies on replacing symbolic object evolution with differentiable vector transformations. The feature representation extracted by the convolutional backbone is first mapped into a shared object space via an affine transformation:

$$o_0 = \tanh(W_{in} f + b_{in}),$$

where $W_{in} \in \mathbb{R}^{D\times 2048}$ is a learnable input weight matrix, $b_{in} \in \mathbb{R}^D$ is a learnable bias vector, and $\tanh(\cdot)$ denotes the element-wise hyperbolic tangent activation function that ensures numerical stability by bounding features within $[-1, 1]$. This resulting vector $o_0 \in \mathbb{R}^D$ is then duplicated across all membranes to initialize the local object configurations, such that $o_0^{(m)} = o_0$ for all $m$. The initial cell states $s_0^{(m)}$ are initialized as a globally shared learnable tensor $W_{init} \in \mathbb{R}^{1\times M\times S}$, which is expanded to match the processing batch size. $W_{init} \in \mathbb{R}^{1\times M\times S}$ means a tensor containing initial states for all $M$ cells and $S$ hidden dimensions and it is broadcast to match batch size during training.

Each membrane cell then performs local rule-based evolution over $T$ discrete time steps. In classical tissue P systems, rules rewrite and transfer symbolic objects between membranes. In the proposed neural formulation, these rules are approximated by parallel sets of learnable linear mappings. For each rule $r \in \{1, \ldots, R\}$, where $R$ is the total number of evolution rules available per cell, the state update and object evolution are defined as follows:

$$s_{t,r}^{(m)} = W_r^{(s)}[s_t^{(m)}; o_t^{(m)}] + b_r^{(s)},$$

$$o_{t,r}^{(m)\prime} = W_r^{(o)}[s_t^{(m)}; o_t^{(m)}] + b_r^{(o)},$$

where $[s_t^{(m)}; o_t^{(m)}] \in \mathbb{R}^{S+D}$ denotes the concatenation of the current cell state and the representation of the local object. The concatenation of $[s_t^{(m)}; o_t^{(m)}]$ means joining two vectors end-to-end into a single vector of size $S + D$. The matrices $W_r^{(s)} \in \mathbb{R}^{S\times(S+D)}$ and $W_r^{(o)} \in \mathbb{R}^{D\times(S+D)}$ represent the learnable transformation weights for the $r$-th rule, while $b_r^{(s)} \in \mathbb{R}^S$ and $b_r^{(o)} \in \mathbb{R}^D$ are their respective biases. Consequently, the application of symbolic rules is generalized to continuous feature evolution, yielding candidate intermediate configurations $s_{t,r}^{(m)}$ and $o_{t,r}^{(m)}$.

Although the discrete rule selection and communication mechanisms of classical tissue P systems are inherently combinatorial, their continuous relaxation in the NeuralTPSLayer replaces every such mechanism with dense, differentiable tensor operations: rule selection is computed as a softmax-weighted combination over all rules, and inter-cell communication is mediated by the fully dense, learnable synapse matrix $W_{syn}$, in which only the diagonal is masked to zero. Consequently, none of the layer's internal computations are sparse or irregular in the sense of classical graph-structured or rule-based P system execution; they consist entirely of batched matrix multiplications, linear projections, and elementwise nonlinearities. Although individual entries of the raw matrix $W_{syn}$ may be negative, the effective communication weights are obtained by Softmax normalization and are therefore non-negative transmission probabilities over a dense $M \times M$ topology (with a zero diagonal). The only structural constraint is

the sequential dependency across the $T$ evolutionary steps, since the cell states and object representations at each step depend on those of the previous one – analogous to the unrolling performed in recurrent neural networks, where a single computational graph is built by looping over $T$ iterations rather than through non-differentiable control flow. Because this sequential loop operates purely on dense tensors residing on the same device as the rest of the network, the NeuralTPSLayer executes entirely on the GPU together with the convolutional feature extractor and the classification head, avoiding any additional host-device synchronization overhead.

At each time step, all transformation rules are evaluated in parallel. Their contributions are modulated through a soft attention mechanism, with attention weights computed as:

$$\alpha_t^{(m)} = \mathrm{Softmax}\big(W_{att}[s_t^{(m)}; o_t^{(m)}] + b_{att}\big),$$

where $W_{att} \in \mathbb{R}^{R\times(S+D)}$ is the attention weight matrix, $b_{att} \in \mathbb{R}^R$ is the attention bias vector, and $\alpha_t^{(m)} \in \mathbb{R}^R$ represents the probability distribution across all $R$ rules. This mechanism acts as a differentiable analog of nondeterministic rule selection in classical membrane systems. Instead of activating a single discrete rule, the model computes a weighted superposition of rule outputs, acting as a continuous relaxation of the maximally parallel rule application mechanism:

$$s_t^{(m)\prime} = \sum_{r=1}^{R} \alpha_{t,r}^{(m)} \cdot s_{t,r}^{(m)},$$

$$o_t^{(m)\prime} = \sum_{r=1}^{R} \alpha_{t,r}^{(m)} \cdot o_{t,r}^{(m)\prime},$$

where $s_t^{(m)\prime} \in \mathbb{R}^S$ and $o_t^{(m)\prime} \in \mathbb{R}^D$ represent the combined local raw states and objects prior to intercellular routing.

Information flow and intercellular communication within the system are regulated by gating mechanisms operating on the evolved configurations:

$$g_{stay}^{(m)} = \sigma\big(W_{stay} s_t^{(m)\prime} + b_{stay}\big),$$

$$g_{go}^{(m)} = \sigma\big(W_{go} s_t^{(m)\prime} + b_{go}\big),$$

where $W_{stay}, W_{go} \in \mathbb{R}^{D\times S}$ are gating weight matrices, $b_{stay}, b_{go} \in \mathbb{R}^D$ are gating biases, and $\sigma(\cdot)$ denotes the element-wise sigmoid function mapping values to the open-close range $[0, 1]$. The objects that remain within the originating cell ($o_{stay}^{(m)} \in \mathbb{R}^D$) and those prepared for intercellular transfer ($o_{go,pool}^{(m)} \in \mathbb{R}^D$) are determined by:

$$o_{stay}^{(m)} = o_t^{(m)\prime} \odot g_{stay}^{(m)},$$

$$o_{go,pool}^{(m)} = o_t^{(m)\prime} \odot (1.0 - g_{stay}^{(m)}) \odot g_{go}^{(m)},$$

where $\odot$ represents the Hadamard (element-wise) product. This interaction ensures that objects allocated for transfer cannot remain static simultaneously within the cell.

Intercellular communication is governed by the learnable synapse matrix $W_{syn} \in \mathbb{R}^{M\times M}$. To strictly prevent self-loops and enforce a valid communication topology in which cells cannot transmit objects to themselves, a strict diagonal masking is applied to the synaptic weights before a Softmax normalization establishes the final directed adjacency matrix $A \in \mathbb{R}^{M\times M}$:

$$A = \mathrm{Softmax}\big(W_{syn} \odot (1.0 - I)\big),$$

where $I \in \mathbb{R}^{M\times M}$ denotes the identity matrix, ensuring that $A_{m,m} = 0$ for all $m$. The total transferred object configuration received by each cell $m$ from its neighboring synapses is then aggregated via batch matrix multiplication:

$$o^{(m)}_{go,received} = \sum_{j=1}^{M} A_{j,m} \cdot o^{(j)}_{go,pool},$$

where $A_{j,m}$ represents the probability of directional transmission from cell $j$ to cell $m$, and $o^{(m)}_{go,received} \in \mathbb{R}^D$ is the total incoming vector configuration.

The final cell states and local object configurations for the next time step $t+1$ are computed using non-linear recurrent updates:

$$s^{(m)}_{t+1} = \tanh(s^{(m)\prime}_t),$$

$$o^{(m)}_{t+1} = \tanh(o^{(m)}_{stay} + o^{(m)}_{go,received}),$$

where the tanh function acts as a continuous bounding mechanism, finalizing the synchronized multi-agent update cycle.

The system evolves iteratively over $T$ discrete time steps, preserving long-range routing dependencies throughout the tissue structure. In the implementation, these $T$ steps are realized by an explicit loop over a shared set of parameters, analogous to RNN unrolling: each iteration appends the corresponding tensor operations to the same computational graph, so reverse-mode automatic differentiation propagates gradients through every step via the chain rule (back-propagation through time) without requiring a separately hand-written back-propagation routine. From a representational perspective, the `NeuralTPSLayer` functions as a structured hidden layer whose internal dynamics approximate the evolution of a tissue membrane system. Unlike standard activation functions that apply independent element-wise nonlinearities, the membrane-inspired layer performs collective state evolution through communication, rule interaction, attention, and recurrence. This enables richer feature interactions and increases the expressive capacity of the hidden representation.

After $T$ iterations, the final tissue representation is obtained by aggregating the cell states across all cells:

$$h_{mem} = \frac{1}{M} \sum_{m=1}^{M} s^{(m)}_T,$$

where $s_T^{(m)}$ is the hidden terminal state of cell $m$ at step $T$, and $h_{mem} \in \mathbb{R}^S$ is the unified multi-cell embedding vector. This aggregation operation reduces sensitivity to local fluctuations in individual membrane states and improves robustness against noisy feature activations. Consequently, the proposed architecture exhibits reduced performance sensitivity compared to conventional feedforward activation layers.

The classification module consists of a single linear layer applied to this representation, producing logits as:

$$z = W_{cls} h_{mem} + b_{cls},$$

where $W_{cls} \in \mathbb{R}^{C \times S}$ is the classification weight matrix, $b_{cls} \in \mathbb{R}^C$ is the classification bias vector, and $z \in \mathbb{R}^C$ represents the raw scoring outputs across classes. These are then converted into class probabilities using the Softmax function:

$$\hat{y} = \text{Softmax}(z),$$

where $\hat{y} \in \mathbb{R}^C$ is the final model prediction vector that contains the estimated probability for each target category.

The complete architecture is fully differentiable and trained end-to-end, allowing gradients to propagate through the linear classifier, the `NeuralTPSLayer` – including all iterative computation steps – and the fine-tuned convolutional backbone. This enables joint optimization of all components within a unified learning framework.

In the context of the evaluated datasets, $S$ denotes the number of dimensions of internal state of the NeuralTPSLayer cells and is independent of the dataset. In contrast, C represents the number of target classes, which varies across datasets. Specifically, for the Oxford-IIIT Pet dataset, $C = 37$ corresponding to 37 cat and dog breeds; for the Caltech-UCSD Birds-200-2011 dataset, $C = 200$ corresponding to 200 bird species; and for the Pl@ntNet-300K dataset, $C = 1081$, corresponding to 1081 plant species.

## 4 Implementation

The proposed implementation is dataset-agnostic and is evaluated on multiple image classification benchmarks described in the experimental section. The output dimensionality of the final classification layer is adjusted according to the number of target classes in the corresponding dataset.

Input images are represented as RGB tensors and resized to $224 \times 224$ pixels to match the expected input resolution of the pretrained ResNet50 backbone. Pixel intensities are normalized using the standard ImageNet channel statistics (mean $\mu = [0.485, 0.456, 0.406]$ and standard deviation $\sigma = [0.229, 0.224, 0.225]$). During training, data augmentation techniques are applied to improve generalization and reduce overfitting, including random resizing, random cropping (scaling between 0.7 and 1.0), random horizontal flipping, and color jittering controlling brightness, contrast, saturation, and hue.

To ensure proper validation and generalization tracking, the raw image directory is structured into a rigorous stratified train-validation-test split (with a 70%/15%/15% proportion), ensuring that each animal breed is proportionally represented in all subsets. This multi-level directory layout is managed by the data pipeline, as illustrated in Figure 2.

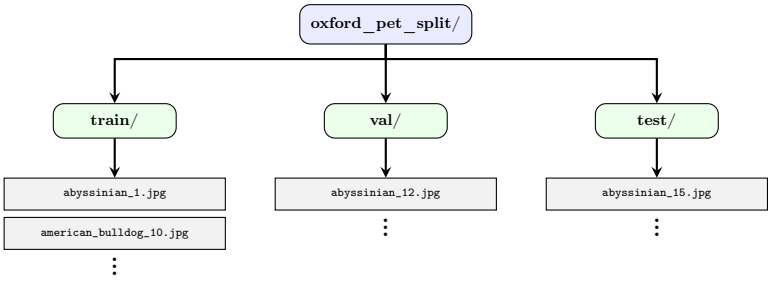



**Fig. 2.** The structured logical layout of the stratified data directories used during training, verification, and final evaluation stages in Oxford-IIIT Pet Dataset split.

Unlike the Oxford–IIIT Pet dataset, where class labels correspond directly to breed names, the Caltech–UCSD Birds-200-2011 dataset organizes samples into fine-grained species-level categories. Within each category, image files adhere to a systematic naming convention that concatenates the species identifier with a unique image index. For example, images belonging to the `Black_Footed_Albatross` category are stored under filenames such as `Black_Footed_Albatross_0001_796111.jpg`, `Black_Footed_Albatross_0002_55.jpg`, and as shown in Figure 3, `Black_Footed_Albatross_0003_796136.jpg`. During preprocessing, class labels are generated automatically from the directory hierarchy, while the individual filenames serve only as unique sample identifiers and are not directly utilized by the neural architecture.

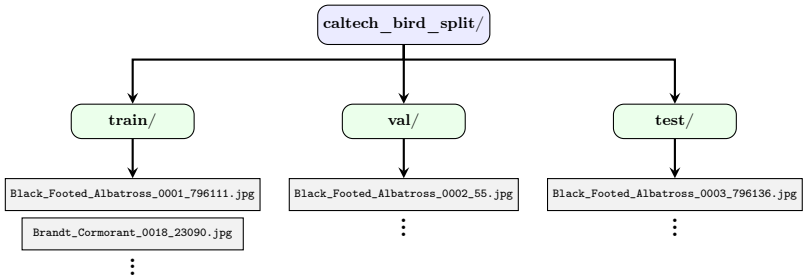



**Fig. 3.** The structured logical layout of the stratified data directories used during training, verification, and final evaluation stages in Caltech-UCSD Birds-200-2011 Dataset split.

The Pl@ntNet-300K dataset follows a different organizational strategy. In the original dataset, plant species are represented by numerically encoded category identifiers rather than descriptive class names. Consequently, the raw image collection does not directly provide human-readable species labels through the file naming convention. To facilitate training and automated label generation, the dataset was reorganized into class-specific directories, where each directory corresponds to a unique numerical category identifier. The images within a given directory were subsequently renamed according to a consistent convention of the form `test_1356111_1.jpg`, `test_1356111_2.jpg`, `train_1356111_1.jpg`, etc. This preprocessing step ensured a uniform directory hierarchy across all evaluated datasets, allowing class labels to be generated automatically from the folder structure while preserving the original category assignments.

The vertical signal flow of the operational network is structurally reshaped compared to feedforward multilayers. The central tissue P system layer acts as a direct replacement for the multi-layer non-linear hidden mappings, establishing a direct topological pipeline mapping the convolutional tensor to low-dimensional linear logits, as shown in Figure 4.

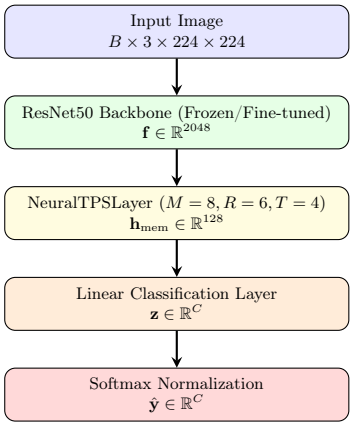



**Fig. 4.** The operational data flow diagram highlighting the direct feed from the `NeuralTPSLayer` to the single linear classification head.

Feature extraction is performed using a pretrained ResNet50 architecture. The final dense classification projection is truncated, allowing the global average pooling layer of the remaining convolutional backbone to directly emit a 2048-dimensional feature representation $f \in \mathbb{R}^{2048}$. To decouple generic semantic feature extraction from target specific fine-tuning and ensure computational efficiency, the lower convolutional layers (specifically up to `layer1`) are strictly frozen, while the remaining deeper residual blocks are left un-frozen to undergo fine-tuning.

The extracted feature vector is then directly routed to the `NeuralTPSLayer`. In the physical implementation, the configuration hyperparameters are:

- number of membrane cells: $M = 8$,
- number of parallel learnable rules per cell: $R = 6$,
- cell hidden state dimensionality: $S = 128$,
- object space dimensionality: $D = 256$,
- total sequential evolution time steps: $T = 4$.

The layer itself follows the equations of Section 3: input projection into object space, attention-weighted rule updates, stay/go gating, dense synapse routing via $W_{syn}$, and $T$-step recurrent evolution ending in mean pooling to $h_{mem}$. All of these operations are realized with standard PyTorch modules (`nn.Linear`, `softmax`, `sigmoid`, `tanh`, and batched tensor algebra). Gradients through the $T$ steps are obtained automatically by reverse-mode automatic differentiation (backpropagation through time), without a hand-written backward pass.

The parallel rule computations do not utilize standalone multi-layer networks; instead, a highly cohesive, unified weight projection architecture is deployed. For each rule $r$, the transformations are parameterized by two unified projection layers that map the concatenated state and object vectors

$[s_t^{(m)}; o_t^{(m)}] \in \mathbb{R}^{S+D}$ to intermediate spaces, generating candidate updates $s_{t,r}^{(m)}$ and $o_{t,r}^{(m)}$ via affine combinations.

Attention weights $\alpha_t^{(m)} \in \mathbb{R}^R$ govern nondeterministic rules by feeding the concatenated state-object representations into an affine attention head, mapping directly to rule scores followed by a Softmax normalization across the rule index. This enables the maximally parallel superposition of candidate objects and candidate states simultaneously.

Intercellular communication is dynamically managed through open-close sigmoid channels. The routing gating values ($g_{stay}^{(m)}, g_{go}^{(m)} \in \mathbb{R}^D$) are projected directly from the rule-aggregate hidden cell state ($s_t^{(m)\prime}$) through element-wise sigmoidal activations. Objects allocated for transfer are masked by the inverted static gate, preventing numerical overlap: $o_{go,pool}^{(m)} = o_t^{(m)\prime} \odot (1.0 - g_{stay}^{(m)}) \odot g_{go}^{(m)}$.

The topology of the cellular network is not static or limited to a simple sequential chain; instead, it utilizes a fully dense, learnable synapse matrix $W_{syn} \in \mathbb{R}^{M \times M}$. Self-loops are explicitly suppressed by applying an item-by-item zeroing mask on the matrix diagonal via an identity matrix subtraction. Normalization across valid incoming links is enforced using a Softmax operation, producing a mathematically valid directed adjacency matrix $A$ where $A_{m,m} = 0$. Transferred object arrays are routed across cells via optimized batch matrix multiplications, removing the necessity for manual iterative graph traversals.

The synchronization update step concludes each evolutionary step by aggregating localized and transferred arrays, followed by a stabilizing bounding activation:

$$s_{t+1}^{(m)} = \tanh(s_t^{(m)\prime}),$$

$$o_{t+1}^{(m)} = \tanh(o_{stay}^{(m)} + o_{go,received}^{(m)}).$$

Upon terminating the evolution loops at step $T$, the global tissue state embedding $h_{mem} \in \mathbb{R}^{128}$ is extracted by averaging across the cell index.

The nonlinear mapping performed by the NeuralTPSLayer arises from the composition of several differentiable operations within each evolution step. The attention-based rule selection produces input-dependent weights for combining the candidate rule transformations and the sigmoid-based gates introduce multiplicative, input-dependent modulation of the information flow. The cell states are subsequently passed through the tanh activation where the resulting states are used as inputs to the next evolution step, repeated over $T$ evolution steps. Thereby, the complete layer implements a recurrent nonlinear transformation obtained by composing the differentiable update functions across the evolution.

In particular, the gradient is propagated backwards through the $T$ evolution steps according to the chain rule. The derivatives of the nonlinear operations, such as $\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$ and $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$, are handled automatically by the PyTorch autograd engine. Gradients are likewise propagated through the attention-weighted rule combinations, the gated information transfer, the inter-cell communication operation, and all learnable linear transformations. communication operation, and all learnable linear transformations.

Therefore, the recurrent evolution of the NeuralTPSLayer is fully differentiable, and gradients can be propagated through all $T$ evolution steps and subsequently to the learnable parameters of the layer and the preceding ResNet50 feature extractor.

The NeuralTPSLayer is constructed entirely from standard differentiable PyTorch modules and tensor operations, including `nn.Linear`, `softmax`, `sigmoid`, `tanh`, tensor multiplication, summation, concatenation, stacking, and `einsum`. During the forward pass, PyTorch's automatic differentiation mechanism dynamically constructs a computational graph containing these operations. When the loss gradient is computed using `loss.backward()`, PyTorch automatically applies reverse-mode automatic differentiation through the complete computational graph.

The classification module completely bypasses deep multilayer structures. The embedding $h_{mem}$ is fed directly into a single linear projection layer: $z = W_{cls}h_{mem} + b_{cls}$, producing the target logits. Cross-entropy loss with a label smoothing factor of 0.1 is applied to enhance robustness against intra-class variations and mislabeled data samples:

$$L_{CE} = -\sum_{c=1}^{C} y_c \log \hat{y}_c.$$

Training uses the AdamW optimizer with mini-batch size $B = 16$, initial learning rate $3 \times 10^{-4}$, weight decay $1 \times 10^{-4}$, and a cosine annealing schedule over 30 epochs. The framework is optimized end-to-end.

## 5 Experimental Results

The proposed NeuralTPS-based architecture was evaluated on three benchmark image classification datasets with substantially different characteristics: the Oxford-IIIT Pet dataset, the Caltech-UCSD Birds-200-2011 dataset, and the Pl@ntNet-300K dataset.

All experiments used a ResNet50 backbone with identical training settings, including the training schedule, optimizer, loss function, and data preprocessing pipeline. In each case, the conventional hidden layer of the classification head was replaced by the proposed NeuralTPSLayer. The models were trained for 30 epochs using cross-entropy loss with label smoothing and the AdamW optimizer.

To assess whether membrane-inspired computations provide advantages over conventional hidden layer transformations, an additional baseline experiment was conducted using a standard multilayer perceptron (MLP) classification head. In this baseline configuration, the NeuralTPSLayer was replaced by a conventional fully connected hidden layer of width 128 with ReLU activation, followed by a linear classifier, while keeping the ResNet50 backbone, optimizer, training schedule, and all other hyperparameters unchanged. This comparison isolates the effect of the proposed membrane-inspired computation from other architectural factors. Table 1 reports the number of trainable parameters in the

classification heads (excluding the shared ResNet50 backbone). The NeuralTPS head contains substantially more parameters than the MLP head; consequently, the comparison is not parameter-matched, and part of the observed gains may arise from increased capacity. Stronger capacity-matched baselines that explicitly combine attention, recurrence, and MLP blocks, as well as systematic ablations over $(M, R, T, S, D)$, are left for future work.

| **Head** | **Oxford** ($C$=37) | **CUB** ($C$=200) | **Pl@ntNet** ($C$=1081) |
|---|---|---|---|
| MLP (hidden=128) | 0.27M | 0.29M | 0.40M |
| NeuralTPS | 1.49M | 1.51M | 1.62M |

**Table 1.** Trainable parameter counts of the classification heads (backbone excluded).

| **Dataset** | **MLP Head (%)** | **NeuralTPS (%)** | **Improvement** |
|---|---|---|---|
| Oxford-IIIT Pet | 90.21 | **92.50** | +2.29 |
| Caltech-UCSD Birds-200-2011 | 68.91 | **68.97** | +0.06 |
| Pl@ntNet-300K | 73.9 | **78.72** | +4.82 |

**Table 2.** Comparison between a conventional MLP classification head and the proposed NeuralTPSLayer.

The proposed NeuralTPSLayer consistently matches or exceeds the MLP baseline, with substantial gains on two datasets. Although both approaches rely on the same convolutional feature extractor, the membrane-inspired architecture achieves a higher classification accuracy in every case.

The greatest improvement is observed on the Pl@ntNet-300K dataset, where the NeuralTPSLayer outperforms the conventional MLP head by 4.82 percentage points. This result suggests that iterative cell interactions and distributed information exchange become increasingly beneficial as dataset complexity, class cardinality, and visual variability increase.

A marginal improvement of 0.06 percentage points is observed on the Caltech-UCSD Birds-200-2011 dataset. Given the magnitude of this difference and the fact that all reported results correspond to a single training run without seed-averaged variance estimates, this gain should be interpreted as essentially within experimental noise rather than as evidence of a clear advantage. In contrast, the +4.82 percentage point improvement on Pl@ntNet-300K is substantially larger; therefore, the two results are not comparable in magnitude, and we refrain from claiming a uniform trend across datasets.

Although the Oxford-IIIT Pet dataset is substantially easier, the proposed architecture still provides a measurable improvement over the conventional MLP baseline. This suggests that the benefits of membrane-inspired computation are not restricted to highly challenging classification tasks but can also contribute to representation learning on relatively balanced datasets.

The Oxford-IIIT Pet dataset yielded the highest accuracy (92.5%), followed by Pl@ntNet-300K (78.72%) and Caltech-UCSD Birds-200-2011 (68.97%). The differences in performance reflect the distinct levels of task difficulty, particularly with respect to fine-grained discrimination, class imbalance, and real-world noise.

To further assess the performance of the proposed architecture, additional experiments were conducted using commonly used vision architectures with a linear classification head. ResNet50, EfficientNet-B0, and ViT-B/16 were evaluated using the same 30-epoch training protocol. Table 3 summarizes the validation accuracy obtained on the evaluated datasets.

| **Dataset** | **ResNet50 + Linear** | **EfficientNet-B0 + Linear** | **ViT-B/16 + Linear** | **ResNet50 + NeuralTPS** |
|---|---|---|---|---|
| Oxford-IIIT Pet | 94.67 | 94.04 | 95.75 | **92.50** |
| CUB-200-2011 | 80.84 | 78.24 | 85.06 | **68.97** |
| Pl@ntNet-300K | 85.41 | 83.36 | 88.90 | **78.72** |

**Table 3.** Validation accuracy (%) of the proposed NeuralTPS-based model and benchmark architectures after 30 training epochs.

The benchmark results provide a broader reference for interpreting the performance of the proposed NeuralTPSLayer. On the Oxford-IIIT Pet dataset, the conventional benchmark architectures achieve a higher validation accuracy than the NeuralTPS-based model, with ViT-B/16 obtaining the highest result. On CUB-200-2011 and Pl@ntNet-300K, the linear-head baselines likewise outperform the current NeuralTPS configuration, which helps contextualize the absolute accuracy of the proposed head.

Table 4 reports validation accuracy throughout training for all three datasets.

| **Epoch** | **Oxford-IIIT Pet (%)** | **CUB-200-2011 (%)** | **Pl@ntNet-300K (%)** |
|---|---|---|---|
| 1 | 31.35 | 7.21 | 47.17 |
| 5 | 72.54 | 23.4 | 69.39 |
| 10 | 83.65 | 44.6 | 74.26 |
| 15 | 90.24 | 57.6 | 76.1 |
| 20 | 91.96 | 60.83 | 77.59 |
| 25 | 92.86 | 63.16 | 78.35 |
| 30 | **92.50** | **68.97** | **78.72** |

**Table 4.** Validation accuracy (%) during training across all evaluated datasets.

Across all datasets, consistent convergence behavior is observed. Oxford-IIIT Pet exhibits faster learning dynamics, whereas Caltech-UCSD Birds-200-2011 and Pl@ntNet-300K show progressively slower optimization. Despite this, no instability or divergence occurs during training, indicating robust optimization characteristics of the proposed architecture.

The slower convergence on Caltech-UCSD Birds-200-2011 can be attributed to its fine-grained nature. Early epochs primarily capture coarse inter-class distinctions, while later epochs allow the NeuralTPSLayer to progressively refine subtle morphological features. Similarly, Pl@ntNet-300K exhibits the slowest convergence due to its high class cardinality, noisy labels, and complex natural image conditions.

To further evaluate the robustness, additional experiments were conducted using multiple image transformations. Besides the original RGB images, three modified input variants were considered:

- **Grayscale**: RGB images were converted to single-channel grayscale and replicated across three channels.
- **Edge**: Edge maps were generated using the Canny edge detector, preserving primarily structural and contour information.
- **Gaussian Blur**: Images were filtered using a Gaussian kernel to reduce high-frequency details while preserving coarse object structure.

All transformed datasets were trained using the same NeuralTPS-based architecture and identical hyperparameter settings. Table 5 summarizes the classification accuracies obtained.

Grayscale experiments indicate that the proposed architecture remains capable of extracting discriminative structural information even when color information is removed. The observed performance reduction is expected, particularly for datasets where pigmentation and color patterns constitute important visual cues.

Edge-based experiments provide additional insight into the role of shape information. Since edge maps largely suppress texture and color information, successful classification on these inputs suggests that the NeuralTPSLayer can effectively exploit object boundaries and geometric structures through its iterative information propagation mechanism.

The Gaussian blur experiments evaluate the robustness against image degradation and loss of fine-grained details. Although high-frequency texture information is attenuated, the architecture continues to operate on coarse object-level representations, providing insight into its ability to maintain stable performance under reduced visual fidelity.

Overall, the results demonstrate that the proposed NeuralTPSLayer does not rely exclusively on a single category of visual cues. Instead, membrane-inspired communication and iterative state refinement mechanisms allow the model to integrate complementary information sources, contributing to improved robustness across multiple image transformations.

| **Dataset** | **RGB** | **Grayscale** | **Edge** | **Gaussian Blur** |
|---|---|---|---|---|
| Oxford-IIIT Pet | 92.50 | 89.79 | 63.96 | 91.33 |
| Caltech-UCSD Birds-200-2011 | 68.97 | 51.32 | 38.23 | 61.1 |
| Pl@ntNet-300K | 78.72 | 71.2 | 58.67 | 76.04 |

**Table 5.** Accuracy comparison under different image transformations.

As expected, removing color information reduces the classification performance across all datasets, since color is a highly discriminative feature. The effect is most pronounced for the Pl@ntNet-300K dataset, where pigmentation and seasonal variation are critical for species identification. In contrast, the Oxford-IIIT Pet dataset shows a smaller degradation, indicating that structural features such as shape and texture remain informative even without color cues.

Despite the reduction in feature richness, the proposed NeuralTPSLayer maintains stable performance, suggesting robustness to input transformations.

Grayscale experiments also provide insight into the type of information captured by the proposed membrane-inspired architecture. Although all datasets exhibit a reduction in accuracy after the removal of color information, the magnitude of this degradation differs considerably. The relatively small drop observed on the Oxford-IIIT Pet dataset suggests that the model can successfully exploit shape, texture, and structural information. In contrast, the larger decrease in the Caltech-UCSD Birds-200-2011 and Pl@ntNet-300K datasets indicates a stronger reliance on color-dependent features, which is consistent with the biological characteristics of bird plumage and plant pigmentation. These observations suggest that the NeuralTPSLayer does not simply memorize color distributions but is capable of integrating multiple complementary visual cues when available.

The grayscale, edge, and Gaussian blur experiments provide additional insight into the robustness of the proposed architecture under different visual representations. Though performance decreases when information is removed or degraded, the NeuralTPSLayer maintains competitive accuracy across all evaluated transformations.

Figure 5 illustrates the impact of the different image transformations on the classification performance for the Oxford-IIIT Pet dataset.

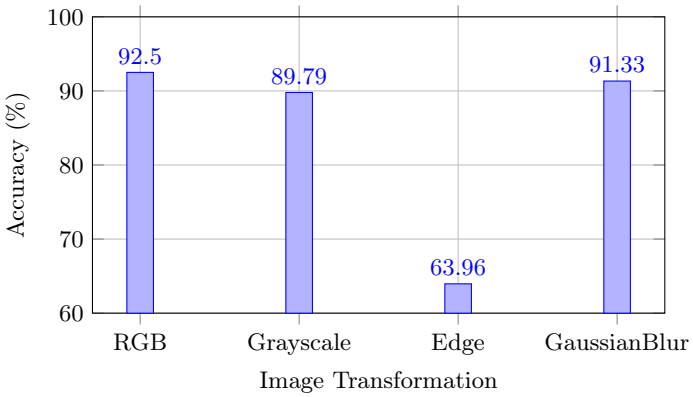



**Fig. 5.** Classification accuracy on the Oxford-IIIT Pet dataset under different image transformations.

The transformation analysis further illustrates the robustness of the proposed NeuralTPSLayer. While performance decreases when visual information is progressively removed or degraded, the architecture maintains relatively high classification accuracy across all evaluated image representations.

The largest reduction is observed for edge-based inputs, where both color and texture information are largely absent. Nevertheless, the model remains capable of exploiting structural and geometric cues, achieving more than 60% accuracy. Gaussian blur causes only a moderate decrease in performance, indicating that the NeuralTPSLayer relies primarily on high-level object representations rather than fine-grained pixel details. These observations support the hypothesis that the membrane-inspired iterative communication mechanism promotes robust feature integration across multiple visual domains.

To visualize learning dynamics, Figure 6 illustrates validation accuracy across epochs for all datasets.

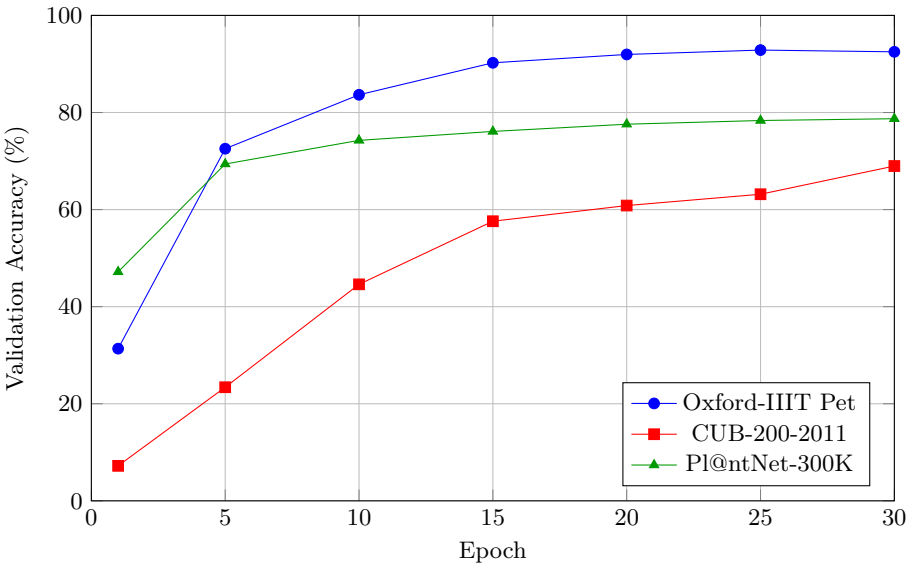



**Fig. 6.** Validation accuracy during training across the three datasets.

The convergence curves reveal an additional characteristic of the proposed architecture. Although the final accuracies differ substantially between datasets, all learning trajectories exhibit a generally increasing trend with minor fluctuations in later epochs. No significant performance degradation or unstable oscillatory behavior is observed during training, indicating stable optimization of the proposed architecture. This suggests that iterative state evolution and attention-guided rule selection mechanisms contribute to stable gradient propagation across training epochs.

Furthermore, the gap between early-epoch and final-epoch performance increases with dataset complexity. On Oxford-IIIT Pet, meaningful representations emerge relatively quickly, whereas the more challenging datasets require substantially longer optimization trajectories. This behavior is consistent with the intended operation of the NeuralTPSLayer, where feature representations are progressively refined through repeated cell interactions and information exchange. As task complexity increases, the benefits of iterative refinement become more apparent, leading to continued performance improvements even during later stages of training.

Another noteworthy observation is that a single NeuralTPSLayer configuration was used across all experiments without dataset-specific modifications. The number of cells, rules, communication mechanisms, and evolution steps remained unchanged despite substantial differences in dataset size and complexity. The ability to maintain competitive performance under these conditions suggests that the proposed membrane-inspired computation framework is relatively robust and does not require extensive architectural tuning when transferred to new image classification domains.

Overall, the results demonstrate that the proposed NeuralTPSLayer can effectively replace conventional hidden layer transformations. While absolute accuracy decreases with increasing dataset complexity, the architecture consistently maintains stable convergence behavior and achieves competitive performance across datasets with varying levels of granularity, imbalance, and visual complexity.

## 6 Conclusions and Outlook

This paper proposes a hybrid neural architecture in which the conventional hidden layer transformation is replaced by a differentiable module inspired by tissue P systems, the `NeuralTPSLayer`. The layer translates structural motifs of tissue membrane systems – interacting cells, rule-driven updates, synapses, and iterative evolution – into a continuous, end-to-end trainable component placed between a ResNet50 feature extractor and a linear classifier.

Compared to classical tissue P systems, the main advantage of the proposed model lies in its differentiability and seamless integration into deep learning pipelines. Traditional P systems are typically discrete and rely on symbolic multiset rewriting, which makes them difficult to combine with large-scale visual feature extractors. The NeuralTPSLayer should therefore be regarded as a continuous, gradient-trainable relaxation of membrane-inspired structure rather than as a formal P system. Unlike spiking neural P systems, the approach avoids explicit temporal spike encoding and event-driven execution [8, 11], operating instead in a continuous state space aligned with standard supervised vision training.

Relative to standard neural architectures, the main difference is the structured internal computation: a cell-based graph with explicit communication pathways, iterative refinement, and rule-level modulation. Empirically, the NeuralTPSLayer matches or exceeds a conventional MLP head on three benchmarks,

with the largest gain on Pl@ntNet-300K (+4.82pp). Gains on Caltech-UCSD Birds-200-2011 are marginal and, given single-run evaluation, should be interpreted cautiously. Robustness experiments under grayscale, edge, and blur transformations indicate that the model does not rely on a single visual cue. Parameter counts show that the NeuralTPS head is substantially larger than the MLP baseline, so part of the observed gains may arise from increased capacity.

Across all evaluated datasets, training remained stable, with gradual improvement and no pronounced oscillatory behavior. A single NeuralTPSLayer configuration ($M$=8, $R$=6, $T$=4, $S$=128, $D$=256) was used without dataset-specific tuning, suggesting that the membrane-inspired inductive bias transfers across tasks of different scale and difficulty. At the same time, the iterative structure increases computational cost relative to feedforward heads, and the interpretability of learned rule interactions remains limited.

Several directions remain open. Stronger capacity-matched baselines that combine attention, recurrence, and MLP blocks, as well as systematic ablations over $(M, R, T, S, D)$ and multi-seed variance reporting, would further clarify the contribution of the layer. Adaptive or sparsified communication topologies, cell-specific object spaces, and visualization of rule activations could bring the model closer to classical tissue P systems while improving efficiency and interpretability. Although the full pipeline already runs on GPU, further engineering of efficient batched membrane updates for larger configurations is a natural next step.

Another direction for further development is a fully GPU-resident implementation. Although the current NeuralTPSLayer already executes its differentiable tensor operations on the GPU, data loading and preprocessing are still performed on the CPU. Future work could therefore investigate GPU-based data preprocessing, augmentation, batching, and memory management to reduce host-device transfers and improve end-to-end throughput. The evolution steps could be further optimized through fused tensor operations and more efficient batched implementations. Such optimizations would be especially important for larger numbers of cells, rules, state dimensions, or evolution steps. A fully GPU-oriented implementation could consequently improve scalability while preserving the differentiable and iterative structure of the proposed model.

In summary, the results suggest that membrane-inspired hidden-layer computation is a viable direction for biologically motivated image classification, while leaving substantial room for tighter baselines, ablations, and closer alignment with formal tissue P systems.

## References

1. Bishop, C.M., Bishop, H.: Deep Learning - Foundations and Concepts. Springer (2024)
2. Chen, H., Liu, X.: Reweighted multi-view clustering with tissue-like P system. PLOS ONE **18**, e0269878 (2023)
3. Christinal, H.A., Díaz-Pernil, D., Real, P.: Region-based segmentation of 2D and 3D images with tissue-like P systems. Pattern Recognit. Lett. **32**(16), 2206–2212 (2011)

4. Díaz-Pernil, D., Gallego-Ortiz, P., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Descriptional complexity of tissue-like P systems with cell division. In: Calude, C.S., Costa, J.F., Dershowitz, N., Freire, E., Rozenberg, G. (eds.) Unconventional Computation, 8th International Conference, UC 2009, Ponta Delgada, Azores, Portugal, September 7-11, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5715, pp. 168–178. Springer (2009)
5. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Solving subset sum in linear time by using tissue P systems with cell division. In: Mira, J., Álvarez, J.R. (eds.) Bio-inspired Modeling of Cognitive Tasks, Second International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2007, La Manga del Mar Menor, Spain, June 18-21, 2007, Proceedings, Part I. Lecture Notes in Computer Science, vol. 4527, pp. 170–179. Springer (2007)
6. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A uniform family of tissue P systems with cell division solving 3-col in a linear time. Theor. Comput. Sci. **404**(1-2), 76–87 (2008)
7. Díaz-Pernil, D., Berciano, A., Peña-Cantillana, F., Gutiérrez-Naranjo, M.A.: Segmenting images with gradient-based edge detection using membrane computing. Pattern Recognition Letters **34**(8), 846–855 (2013)
8. Ermini, I., Zandron, C.: Modular spiking neural membrane systems for image classification. Int. J. Neural Syst. **34**(6), 2450021:1–2450021:16 (2024)
9. Freund, R., Păun, G., Pérez-Jiménez, M.J.: Tissue P systems with channel states. Theor. Comput. Sci. **330**(1), 101–116 (2005)
10. Garcin, C., Joly, A., Bonnet, P., Lombardo, J.C., Affouard, A., Chouet, M., Servajean, M., Lorieul, T., Salmon, J.: Pl@ntNet-300K: A plant image dataset with high label ambiguity and a long-tailed distribution. In: NeurIPS Datasets and Benchmarks 2021 (2021)
11. Gatti, M., Barbato, J., Zandron, C.: Spiking neural network classification of X-ray chest images. Knowledge-Based Systems **314**, 113194 (2025)
12. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
13. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
14. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundam. Inform. **71**(2-3), 279–308 (2006)
15. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**, 436–444 (2015)
16. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Characterising the complexity of tissue P systems with fission rules. J. Comput. Syst. Sci. **90**, 115–128 (2017)
17. Macías-Ramos, L.F., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rius-Font, M., Valencia-Cabrera, L.: The efficiency of tissue P systems with cell separation relies on the environment. In: Membrane Computing - 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7762, pp. 243–256. Springer (2012)
18. Martín-Vide, C., Păun, G., Pazos, J., Rodríguez-Patón, A.: Tissue P systems. Theor. Comput. Sci. **296**(2), 295–326 (2003)
19. Orellana-Martín, D., Valencia-Cabrera, L., Song, B., Pan, L., Pérez-Jiménez, M.J.: Tissue P systems with evolutional communication rules with two objects in the left-hand side. Nat. Comput. **22**(1), 119–132 (2023)
20. Pan, L., Pérez-Jiménez, M.J.: Computational complexity of tissue-like P systems. J. Complex. **26**(3), 296–315 (2010)

21. Parkhi, O.M., Vedaldi, A., Zisserman, A., Jawahar, C.V.: Cats and dogs. In: IEEE Conference on Computer Vision and Pattern Recognition (2012)
22. Peng, H., Wang, J., Shi, P., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Fault diagnosis of power systems using fuzzy tissue-like P systems. Integr. Comput. Aided Eng. **24**(4), 401–411 (2017)
23. Pérez-Jiménez, M.J., Sosík, P.: An optimal frontier of the efficiency of tissue P systems with cell separation. Fundam. Informaticae **138**(1-2), 45–60 (2015)
24. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000)
25. Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Tissue P systems with cell division. Int. J. Comput. Commun. Control **3**(3), 295–303 (2008)
26. Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press (12 2009)
27. Song, B., Li, K., Orellana-Martín, D., Pérez-Jiménez, M.J., Pérez-Hurtado, I.: A survey of nature-inspired computing: Membrane computing. ACM Comput. Surv. **54**(1), 22:1–22:31 (2022)
28. Song, B., Pan, L.: The computational power of tissue-like P systems with promoters. Theor. Comput. Sci. **641**, 43–52 (2016)
29. Song, B., Song, T., Pan, L.: A time-free uniform solution to subset sum problem by tissue P systems with cell division. Math. Struct. Comput. Sci. **27**(1), 17–32 (2017)
30. Song, B., Zhang, C., Pan, L.: Tissue-like P systems with evolutional symport/antiport rules. Inf. Sci. **378**, 177–193 (2017)
31. Sosík, P., Cienciala, L.: A limitation of cell division in tissue P systems by PSPACE. Journal of Computer and System Sciences **81**(2), 473–484 (2015)
32. Sosík, P., Paul, P., Ciencialová, L.: A survey on learning models of spiking neural membrane systems. Natural Computing **24**, 665–677 (2025)
33. Wah, C., Branson, S., Welinder, P., Perona, P., Belongie, S.: Caltech-UCSD Birds-200-2011. Tech. Rep. CNS-TR-2011-001, California Institute of Technology (2011)
34. Wang, L., Liu, X., Qu, J., Zhao, Y., Gao, L., Ren, Q.: An extended membrane system with monodirectional tissue-like P systems and enhanced particle swarm optimization for data clustering. Applied Sciences **13**(13) (2023)
35. Wang, L., Liu, X., Qu, J., Zhao, Y., Jiang, Z., Wang, N.: An extended tissue-like P system based on membrane systems and quantum-behaved particle swarm optimization for image segmentation. Processes **10**(2) (2022)
36. Yahya, R., Shamsuddin, S., Yahya, S., Hasan, S., Alsalibi, B.: Automatic 2D image segmentation using tissue-like P system. International Journal of Advances in Soft Computing and Its Applications **10**(1), 36–54 (2018)
37. Yahya, S., Yahya, R., Alsalibi, B., Al-Khafaji, G., Shamsuddin, S.M.: Three-dimensional image segmentation using tissue-like P system. ARO-The Scientific Journal of Koya University **5**(2), 67–74 (2017)
38. Zhang, X., Luo, B., Pan, L.: Small universal tissue P systems with symport/antiport rules. Int. J. Comput. Commun. Control **7**(1), 173–183 (2012)

# Solving 3-SAT via Enzymatic Numerical P Systems with Membrane Division

Alberto Leporati[0000−0002−8105−4371]

Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)
Università degli Studi di Milano-Bicocca
Viale Sarca 336, Edificio U14, 20126 Milano (MI), Italy
alberto.leporati@unimib.it
https://www.unimib.it/alberto-ottavio-leporati

**Abstract.** While enzymatic numerical (EN) P systems have been extensively studied as mobile robot controller devices and from the computability point of view, less has been done in using them to solve computationally hard problems. Albeit it is known that EN P systems that use the arithmetic operations $+$, $-$, $\times$, $\div$ are able to solve all PSPACE problems in polynomial time, no explicit construction for solving even NP-complete problems has been proposed in the literature. In this paper, we propose both a semi-uniform and a uniform solution of the NP-complete problem 3-SAT by means of EN P systems with (elementary) membrane division, working in the all-parallel mode. Given that standard EN P systems working in all-parallel mode can be "flattened" to single-membrane systems, we discuss the possibility of extending the flattening procedure to the new model of EN P system systems leading to the definition of EN P systems with program and variable duplication. Determining the precise computational power of all these newly introduced models of EN P systems constitutes an interesting open problem.

**Keywords:** 3-SAT problem · Enzymatic numerical P systems · Membrane division · Flattening · Program duplication.

## 1 Introduction

Membrane systems, commonly known as *P systems*, are biologically inspired models of distributed and parallel computing, introduced by Gh. Păun in 1998. Since his seminal work [15], previously circulated as a report of the Turku Centre for Computer Science (TUCS), several variants of membrane systems have been defined, mainly divided in cell-based [15,14], tissue-based [12] and neural-based [8] P systems. Most of these variants have been proven to work as Turing universal number generating/accepting devices, function computing devices, or language generating devices. Furthermore, several kinds of P systems have been used to model biological phenomena or to solve computationally hard optimization problems. For a comprehensive introduction to the field of membrane computing, including the state of the art concerning computability and computational complexity results updated to 2010, we refer the reader to [21].

Numerical P systems have been introduced in [20] as a model of membrane systems inspired both from the structure of living cells and from economics. Each region of a numerical P system contains some numerical variables, whose values evolve by means of *programs*. In [20] it is proved that nondeterministic numerical P systems (working in the sequential way) with polynomial production functions characterize the recursively enumerable sets of natural numbers, while deterministic numerical P systems, with polynomial production functions having non-negative coefficients, compute strictly more than semilinear sets of natural numbers.

Enzymatic numerical P systems (EN P systems, for short) have been introduced in [16] as an extension of numerical P systems in which some variables, named the *enzymes*, control the application of the programs; their most promising application seems to be the simulation of control mechanisms of mobile and autonomous robots, as shown in [3,17,18]. In [10], it is shown that EN P systems working in the all-parallel and in the one-parallel modes characterize $\mathbb{N}\mathbf{RE}$ (the family of recursively enumerable sets of natural numbers) by using just one membrane, and linear production functions that use each at most one variable. As a byproduct, three small universal EN P systems are built: (1) an all-parallel EN P system of degree 1, having 31 variables and 61 programs; (2) a one-parallel deterministic EN P system of degree 1, having 146 variables and 105 programs; (3) a one-parallel deterministic EN P system of degree 1, having 137 variables and 108 programs. These last results have been subsequently improved by many authors (see, for example, [11]), defining increasingly smaller universal EN P systems, in terms of the number of enzymatic variables or number of (linear or quadratic) programs.

From the point of view of using enzymatic numerical P systems to solve difficult computational problems, in [9] it is shown that all-parallel enzymatic numerical P systems whose production functions can be expressed as a combination of sums, differences, products and integer divisions characterise PSPACE when working in polynomial time. It is also shown that, when only sums and differences are available, exactly the problems in the complexity class P can be solved in polynomial time. These results are proved by showing how EN P systems and random access machines, running in polynomial time and using the same basic operations, can simulate each other efficiently.

While this result clearly establishes the limits of the computational efficiency of EN P systems, the number of explicit solutions to computationally hard problems by means of this model published in the literature is very low. Indeed, to the best knowledge of the author only one work [23] exists in the literature that uses EN P systems to solve the NP-complete problem Subset Sum. There, the authors define membrane division programs for EN P systems, that allow them to produce an exponential number of membranes – one for each possible subset of the instance values – in a linear number of time steps. Each membrane then computes the sum of its elements and compares it with the required sum value, setting a variable in the skin membrane if the two values coincide. The proposed solution thus follows the schema introduced by Gh. Păun in [19] to

attack NP-complete problems by P systems with active membranes, and since then become a standard method for solving computationally difficult problems. The solution proposed in [23] is *semi-uniform*, since the definition of the EN P system depends on the instances of SUBSET SUM to be solved. Precisely, each EN P system is able to solve one instance $(\{v_1, v_2, \ldots, v_n\}, S)$ of SUBSET SUM having $n$ values in the set and a desired value for their sum.

Following [23], in this paper we propose both a semi-uniform and a uniform family of EN P systems with membrane division that solve the 3-SAT problem. In the semi-uniform family, each system solves exactly one instance of 3-SAT, whereas in the uniform family each system is able to solve any instance of 3-SAT defined on $n$ Boolean variables. The proposed systems work in the all-parallel mode. By exploiting the fact that such systems can be "flattened" to a single-membrane, we discuss an equivalent model of EN P systems whose programs and variables can be duplicated, each producing an exponential number of copies, during the computation.

The rest of the paper is organized as follows. In Section 2 we recall some basic definitions, comprising the 3-SAT problem, the operation of EN P systems, and the working of random access machines (RAMs). In Section 3 we extend EN P systems by introducing membrane division programs, and we describe how such systems can solve a fixed instance of 3-SAT. In Section 4 we then describe a uniform family of EN P systems with division for elementary membranes, where each P system of the family solves in polynomial time any instance of 3-SAT defined over $n$ Boolean variables. In Section 5 we discuss the flattening of the proposed systems onto a single membrane, and their relationship with a new model of EN P systems in which programs and variables can be duplicated during the computation. Finally, Section 6 concludes the paper and provides some directions for future work.

## 2 Mathematical Preliminaries

$\mathbb{N} = \{0, 1, 2, \ldots\}$ denotes the set of non-negative integers. An alphabet $\Sigma$ is a finite and non-empty set of elements, called *symbols*.

A Boolean variable $x$ is a variable that can take two different values, either 1 or 0, that can be interpreted as the logical values *true* and *false*, respectively. A literal is either a Boolean variable $x$ or the negation of a Boolean variable $\neg x$. A clause is a disjunction of literals. A propositional Boolean formula in conjunctive normal form (CNF) is a conjunction of clauses. Let $Var(\phi) = \{x_1, \ldots, x_n\}$ be the set of variables appearing in a propositional Boolean formula $\phi$; then, a truth assignment $\sigma$ to such a formula $\phi$ is a function $\sigma : Var(\phi) \to \{1, 0\}$ that assigns a truth value to each variable. The number of all possible assignments to the variables of $\{x_1, \ldots, x_n\}$ is $2^n$. Once each Boolean variable has been assigned a truth value, clauses and formulas can be evaluated by interpreting the logical operators in the usual way. We thus obtain that a clause is satisfied if any of its literals is satisfied, whereas a CNF formula is satisfied if all of its clauses are satisfied.

3-SAT is an NP-complete decision problem [6] that can be formulated as follows: given a CNF Boolean formula $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$, where each clause $C_j = X_{j_1} \vee X_{j_2} \vee X_{j_3}$ is the disjunction of exactly three literals $X_{j_1}, X_{j_2}, X_{j_3}$ defined over the set of variables $Var(\phi) = \{x_1, \ldots, x_n\}$, determine whether there exists an assignment $\sigma : Var(\phi) \to \{1, 0\}$ that satisfies $\phi$. In what follows, we write $x_i \in C_j$ (respectively, $\neg x_i \in C_j$) to indicate that the literal $x_i$ (resp., $\neg x_i$) appears in the clause $C_j$. Without loss of generality, we can avoid the clauses in which the same literal is repeated or both the literals $x_i$ and $\neg x_i$, for any $1 \leq i \leq n$, occur. As a consequence, each clause of a 3-SAT instance can be obtained by considering a subset of $\{x_1, \ldots, x_n\}$ containing exactly three variables, each being subsequently negated or not. With a little abuse of notation, we will denote by 3-SAT$(n)$ the set of instances of 3-SAT which have $n$ variables. By the above reasoning, it is not difficult to see that the number of possible clauses of 3-SAT$(n)$ is $8 \cdot \binom{n}{3}$, which is proportional to $n^3$.

In this paper we will first show how to solve any instance of 3-SAT$(n)$ in a semi-uniform way by EN P systems empowered with membrane division programs. Then, we will build a uniform family $\mathbf{\Pi} = \{\Pi_{\text{3-SAT}}(n)\}_{n \in \mathbb{N}}$ of EN P systems such that for all $n \in \mathbb{N}$ the system $\Pi_{\text{3-SAT}}(n)$ solves all the instances of 3-SAT$(n)$ in a number of computation steps which is polynomial in $n$. All the systems will work in the all-parallel mode, and hence will be deterministic, as explained in the next section.

### 2.1 Enzymatic Numerical P Systems

An *enzymatic numerical P system* is a construct of the form

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0))),$$

where $m \geq 1$ is the degree of the system (the number of membranes), $H$ is an alphabet of labels, $\mu$ is a tree-like membrane structure with $m$ membranes injectively labeled with elements of $H$, $Var_i$ and $Pr_i$ are respectively the set of variables and the set of programs that reside in region $i$, and $Var_i(0)$ is the vector of initial values for the variables of $Var_i$. All sets $Var_i$ and $Pr_i$ are finite. In the original definition of EN P systems [16] the values assumed by the variables may be real, rational or integer numbers; in what follows we will allow instead only integer numbers. The variables from $Var_i$ are written in the form $x_{j,i}$, for $j$ running from 1 to $|Var_i|$, the cardinality of $Var_i$; the value assumed by $x_{j,i}$ at time $t \in \mathbb{N}$ is denoted by $x_{j,i}(t)$. Similarly, the programs from $Pr_i$ are written in the form $P_{l,i}$, for $l$ running from 1 to $|Pr_i|$.

The programs allow the system to change the values of variables during computations. Each program consists of two parts: a *production function* and a *repartition protocol*. The former can be any function using variables from the region that contains the program. Usually only polynomial functions are considered, since these are sufficient to reach the computational power of Turing machines, as proved in [25]. Using the production function, the system computes a *production value* from the values of its variables at that time. This value

is distributed to variables from the region where the program resides, and to variables in its upper (parent) and lower (children) compartments, as specified by the repartition protocol. Formally, for a given region $i$, let $v_1, \ldots, v_{n_i}$ be all these variables; let $x_{1,i}, \ldots, x_{k_i,i}$ be some variables from $Var_i$, let $F_{l,i}(x_{1,i}, \ldots, x_{k_i,i})$ be the production function of a given program $P_{l,i} \in Pr_i$, and let $c_{l,1}, \ldots, c_{l,n_i}$ be natural numbers. The program $P_{l,i}$ is written in the following form:

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}) \to c_{l,1}|v_1 + c_{l,2}|v_2 + \cdots + c_{l,n_i}|v_{n_i}, \tag{1}$$

where the arrow separates the production function from the repartition protocol. Let $C_{l,i} = \sum_{s=1}^{n_i} c_{l,s}$ be the sum of all the coefficients that occur in the repartition protocol. If the system applies program $P_{l,i}$ at time $t \geq 0$, it computes the value

$$q = \frac{F_{l,i}(x_{1,i}(t), \ldots, x_{k_i,i}(t))}{C_{l,i}},$$

that represents the "unitary portion" to be distributed to variables $v_1, \ldots, v_{n_i}$ proportionally with coefficients $c_{l,1}, \ldots, c_{l,n_i}$. So each of the variables $v_s$, for $1 \leq s \leq n_i$, will receive the amount $q \cdot c_{l,s}$. An important observation is that variables $x_{1,i}, \ldots, x_{k_i,i}$ involved in the production function are reset to zero after computing the production value, while the other variables from $Var_i$ retain their value. The quantities assigned to each variable from the repartition protocol are added to the current value of these variables, starting with 0 for the variables which were reset by a production function. As pointed out in [25], a delicate problem concerns the issue whether the production value is divisible by the total sum of coefficients $C_{l,i}$. As it is done in [25], in this paper we assume that this is always the case by construction, so we will deal only with such systems; see [20] for other possible approaches.

Besides programs of type (1), EN P systems may also have programs of the form

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i})|_{e_{j,i}} \to c_{l,1}|v_1 + c_{l,2}|v_2 + \cdots + c_{l,n_i}|v_{n_i},$$

where $e_{j,i}$ is a variable from $Var_i$ different from $x_{1,i}, \ldots, x_{k_i,i}$ and from $v_1, \ldots, v_{n_i}$. Such a program can be applied at time $t$ only if $e_{j,i}(t) > \min(x_{1,i}(t), \ldots, x_{k_i,i}(t))$. Variable $e_{j,i}$ operates like an *enzyme*, that enables the execution of the program, but – as it also happens with catalysts – it is neither consumed nor modified by the execution of the program. However, in EN P systems, the enzymes can evolve by means of other programs, that is, they can receive "contributions" from other programs and regions.

A *configuration* of $\Pi$ at time $t \in \mathbb{N}$ is given by the values of all the variables of $\Pi$ at that time; in a compact notation, we can write it as the sequence $(Var_1(t), \ldots, Var_m(t))$. The *initial configuration* can thus be described as the sequence $(Var_1(0), \ldots, Var_m(0))$. The system $\Pi$ evolves from a configuration to another one by means of *computation steps*, in which one or more programs of $\Pi$ are executed according to a *mode* of computation. In [25], at each computation step the programs to be executed are chosen in the so called *sequential* mode: one program is nondeterministically chosen in each region, among the programs that

can be executed at that time. Another possibility is to select the programs in the so called *all-parallel* mode: in each region, all the programs that can be executed are selected, with each variable participating in all programs where it appears. Note that in this case EN P systems become *deterministic*, since nondeterministic choices between programs never occur. A variant of parallelism, analogous to the maximal one which is often used in membrane computing, is the so called *one-parallel* mode: in each region, all the programs which can be executed can be selected, but the actual selection is made in such a way that each variable participates in only one of the chosen programs. We say that the system reaches a *final configuration* if and when it happens that no applicable set of programs produces a change in the current configuration. In such a case, a specified set of variables contains the output of the computation. Of course, a computation may never reach a final configuration. Note that in the usual definition of EN P systems the output of a computation is instead defined as the collection of values taken by a specified set of variables during the whole computation.

EN P systems can be used to compute functions, in the so called *computing mode*, by considering some *input variables* and *output variables*. The initial values of the input variables are interpreted as the actual arguments of the function, while the value of the output variables in the final configuration (provided that the system reaches it) is viewed as the output of the computed function. If the system never reaches a final configuration, then the computed function is undefined for the specified input values. By neglecting input variables, (nondeterministic) EN P systems can also be used in the *generating mode*, whereas by neglecting output variables we can use (deterministic or nondeterministic) EN P systems in the *accepting mode*, where the input is accepted if the system reaches a final configuration.

When using EN P systems in the generating or accepting modes, we denote by $\mathbf{ENP}_m(poly^n(r), app\_mode)$ the family of sets of (vectors of) non-negative integer numbers which are computed by EN P systems of degree at most $m \geq 1$, using polynomials of degree at most $n \geq 0$ with at most $r \geq 0$ arguments as production functions; the fact that the programs are applied in the sequential, one-parallel, or all-parallel mode is denoted by assigning the value *seq*, *oneP* or *allP* to the *app_ mode* parameter, respectively. If one of the parameters $m$, $n$, $r$ is not bounded by a constant value, we replace it by $*$.

Some results concerning the computational power of EN P systems are reported in [25,24]. In particular, the following characterizations of $\mathbb{N}\mathbf{RE}$ (the family of recursively enumerable sets of natural numbers) are proved:

$$\begin{aligned}\mathbb{N}\mathbf{RE} &= \mathbf{ENP}_7(poly^5(5), seq) = \mathbf{ENP}_*(poly^1(2), oneP)\\ &= \mathbf{ENP}_4(poly^1(6), allP).\end{aligned}$$

In [10] the results concerning EN P systems working in the all-parallel and in the one-parallel modes have been improved: in both cases, characterizations of $\mathbb{N}\mathbf{RE}$ are obtained by using just one membrane, and linear production functions that use each at most one variable:

$$\mathbb{N}\mathbf{RE} = \mathbf{ENP}_1(poly^1(1), allP) = \mathbf{ENP}_1(poly^1(1), oneP).$$

EN P systems can also be used as (polynomial) time-bounded recognising devices as follows. Notice that we use two variables (instead of just one of them), named *accept* and *reject*, to signal the end of computations. This is done because some programs of the system may be applied forever, causing the system to never halt even if the configuration does not change any more. By using two variables, the event of reaching a final configuration is made visible and distinguishable from the outside.

**Definition 1.** *Let $L \subseteq \{0,1\}^\star$ be a language, and let $\Pi$ be a deterministic EN P system with two distinguished variables accept and reject. We say that $\Pi$ decides $L$ in polynomial time iff, for all $x \in \{0,1\}^\star$, when the integer having binary representation $1x$ is initially given to a specified input variable*[1] *the P system $\Pi$ reaches a final configuration such that*

- *if $x \in L$, then accept $= 1$ and reject $= 0$*
- *if $x \notin L$, then accept $= 0$ and reject $= 1$*

*within a number of steps bounded by $O(|x|^k)$ for some $k \in \mathbb{N}$.*

Finally, in what follows we will exploit the fact that all-parallel and one-parallel EN P systems can be "flattened", that is, reduced to single-membrane systems. Due to the simplicity of its proof, we entirely recall here the theorem proved in [10].

**Theorem 1.** *Let $\Pi$ be any computing (or generating, or accepting) EN P system of degree $m \geq 1$, working in the all-parallel or in the one-parallel mode. Then there exists an EN P system $\Pi'$ of degree 1 that computes (respectively, generates, accepts) the same function (respectively, family of sets) using the same program application mode.*

*Proof.* Consider an EN P system

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0))).$$

Note that each variable $x_{j,i} \in Var_i$ and each program $P_{l,i} \in Pr_i$ already indicates in one of its indexes the region that contains it. We build a new EN P system $\Pi'$ of degree 1, by putting all the variables and all the programs of $\Pi$ – keeping both indexes, also in the variables occurring in programs – in the membrane of $\Pi'$. Clearly, this establishes a bijection between the variables (resp., programs) of $\Pi$ and the corresponding variables (resp., programs) of $\Pi'$. So any program $P_{l,i}$ of $\Pi$ still operates on the correct variables when transformed and put into $\Pi'$, regardless of whether or not it uses an enzyme. Input and output variables are also preserved. If $\Pi$ works in the all-parallel mode then at each computation step all the programs that can be executed are selected, and the same happens in $\Pi'$ by letting it work in the all-parallel mode. The same applies when $\Pi$ and $\Pi'$ work in the one-parallel mode, so the claim of the theorem follows. □

[1] The "1" is prefixed to the input string $x$ in order to keep the leading zeroes.

### 2.2 Random Access Machines

The proofs in this paper will be based on random access machines [13,7]. We define the specific variant we will employ:

**Definition 2 (RAM).** *A random access machine consists of an infinite number of* registers $(r_i : i \in \mathbb{N})$ *having values in* $\mathbb{N}$*, initially set to zero, and a finite sequence of* instructions *injectively labelled by elements* $\ell \in \mathbb{N}$*. The instructions are of the following types:*

- *assignment of a constant* $k \in \mathbb{N}$*: "*$\ell$: $r_i \leftarrow k$*" (*$r_i$ *is assigned a constant value)*
- *copying a register: "*$\ell$: $r_i \leftarrow r_j$*" (*$r_i$ *is assigned the content of a fixed register)*
- *arithmetic operations, with* $\bullet \in \{+, -, \times, \div\}$*: "*$\ell$: $r_i \leftarrow r_j \bullet r_k$*"*
- *conditional jump, with* $\ell_1, \ell_2 \in \mathbb{N}$*: "*$\ell$: if $r_i \neq 0$ then $\ell_1$ else $\ell_2$*"*
- *halt and accept: "*$\ell$: accept*"*
- *halt and reject: "*$\ell$: reject*".*

The labels of the instructions will sometimes be left implicit.

We assume, without loss of generality, that it is never the case that a register or a label are mentioned multiple times in the same instruction (e.g., in "$\ell : r_i \leftarrow r_j \bullet r_k$" we assume $i \neq j$, $j \neq k$, and $i \neq k$).

Since RAMs operate on natural numbers, we only allow *non-negative* subtraction, i.e., $x - y = 0$ when $y > x$.

**Definition 3.** *Let* $L \subseteq \{0,1\}^\star$ *be a language, and let* $M$ *be a* RAM*. We say that* $M$ decides $L$ *in polynomial time iff, for all* $x \in \{0,1\}^\star$*, when the integer having binary representation* $1x$ *is loaded into a specified input register, the machine* $M$ *behaves as follows:*

- *if* $x \in L$*, then* $M$ *reaches an "*accept*" instruction*
- *if* $x \notin L$*, then* $M$ *reaches a "*reject*" instruction*

*within a number of steps bounded by* $O(|x|^k)$ *for some* $k \in \mathbb{N}$*.*

In [9] it is proved that RAMs can efficiently simulate all-parallel EN P systems, and vice versa, where efficiently means that for each simulation a constant slowdown applies. Here we recall the simulation of RAMs by EN P systems working in the all-parallel mode, as it will be useful in the following. Note that each RAM instruction is simulated in just one step.

**Theorem 2.** *Let* $M$ *be a* RAM*. Then, for each instruction of* $M$ *there exists a set of programs for an all-parallel EN P system* $\Pi$ *that simulates it in one computation step.*

*Proof.* We proceed by examining all possible cases. In what follows, $z$ is a variable whose value is always zero, variables $r_i$, $r_j$, $r_k$ represent registers of $M$ (containing non negative integer values), and variables $p_\ell$ assume values in $\{0,1\}$ to indicate the next instruction of $M$ to be simulated.

RAM instructions of type "$\ell\colon r_i \leftarrow k$" can be simulated by the following set of all-parallel programs:

$$0r_i + k + z|_{p_\ell} \to 1|r_i$$
$$p_\ell \to 1|p_{\ell+1}$$

When $p_\ell = 0$ the first program is not executed, while the second program zeroes $p_\ell$ (thus leaving its value unaltered) and gives a contribution of zero to variable $p_{\ell+1}$, thus behaving as a NOP (No OPeration). Hence no interference is produced in the variables involved in the RAM instruction currently simulated. On the other hand, if $p_\ell = 1$ then the first program first zeroes $r_i$ and then assigns the value $k$ to it, while the second program zeroes $p_\ell$ and sets $p_{\ell+1}$ to 1, thus pointing to the next instruction of $M$ to be simulated.

Assignment instructions of type "$\ell\colon r_i \leftarrow r_j$", with $j \neq i$, can be simulated using the following programs:

$$0r_i + 2r_j + z|_{p_\ell} \to 1|r_i + 1|r_j$$
$$p_\ell \to 1|p_{\ell+1}$$

As in the previous case, when $p_\ell = 0$ the first program is not active while the second one operates like a NOP. When $p_\ell = 1$, instead, the first program first zeroes both $r_i$ and $r_j$ and then assigns to them the old value of $r_j$; the second program, as before, passes the control to instruction $\ell + 1$. Albeit in our definition of RAMs we have avoided the case when $j = i$, here we just observe that we can also easily deal with it: we simply remove the first program, since in this case it always operates like a NOP.

Arithmetic instructions of type "$\ell\colon r_i \leftarrow r_j \bullet r_k$", with $\bullet \in \{+, -, \times, \div\}$ and $i \neq j$, $j \neq k$, and $i \neq k$, can be simulated as follows:

$$0r_i + r_j \bullet r_k + z|_{p_\ell} \to 1|r_i$$
$$r_j + z|_{p_\ell} \to 1|r_j \tag{2}$$
$$r_k + z|_{p_\ell} \to 1|r_k \tag{3}$$
$$p_\ell \to 1|p_{\ell+1}$$

When $p_\ell = 0$ the first three programs are not executed, while the last program behaves as a NOP. On the other hand, if $p_\ell = 1$ then the first program first zeroes variables $r_i$, $r_j$ and $r_k$, and then it assigns to $r_i$ the result of the operation $r_j \bullet r_k$, using the old values of $r_j$ and $r_k$. Programs (2) and (3) are used to preserve the old values of variables $r_j$ and $r_k$, whereas the last program passes the control to instruction $\ell + 1$.

Finally, instructions of type "$\ell\colon$ if $r_i \neq 0$ then $\ell_1$ else $\ell_2$", with $\ell \neq \ell_1$, $\ell \neq \ell_2$, and $\ell_1 \neq \ell_2$, can be simulated by the following programs:

$$p_\ell \to 1|p_{\ell_1}$$
$$r_i - 1|_{p_\ell} \to 1|p_{\ell_1} \tag{4}$$
$$r_i + 1|_{p_\ell} \to 1|p_{\ell_2} \tag{5}$$

in which we assume $r_i \neq 0$ and correct if this is not the case. Note, in particular, that programs (4) and (5) are active if and only if $p_\ell = 1$ and $r_i = 0$. So, when $p_\ell = 0$ only the first program is executed, behaving as a NOP. When $p_\ell = 1$ and $r_i > 0$, the first program passes the control to instruction $\ell_1$ whereas the other two programs are not executed. Finally, when $p_\ell = 1$ and $r_i = 0$ the first program zeroes $p_\ell$ and (incorrectly) sets $p_{\ell_1}$ to 1. This time, however, also the other two programs are executed: after resetting once again the value of $r_i$ to 0, program (4) gives a contribution of $-1$ to $p_{\ell_1}$, so that its final value will be zero, whereas program (5) sets $p_{\ell_2}$ to 1, indicating the next instruction of $M$ to be simulated. □

The computational power of polynomial-time RAMs is strictly dependent on the set of basic operations that can be computed in a single time step. When only addition and subtraction are available, then polynomial-time RAMs are equivalent to polynomial-time Turing machines, and hence characterize the complexity class P [4]. On the other hand, multiplication and division considerably increase the efficiency of polynomial-time RAMs, allowing them to characterize the entire PSPACE [2]. Due to the efficient simulations between the two models, these characterizations extend to all-parallel EN P systems: with only addition and subtraction, this model characterizes P when operating in polynomial time, whereas adding also multiplication and integer division allows it to characterize PSPACE.

## 3 A Semi-Uniform Solution of 3-SAT by EN P Systems with Elementary Membrane Division

As stated above, our first aim is to build a semi-uniform family $\mathbf{\Pi}$ of EN P systems such that each P system in the family solves a specific instance of 3-SAT($n$) in a number of computation steps which is polynomial in $n$. All the systems of $\mathbf{\Pi}$ will be deterministic, operating in the all-parallel mode. In the next section we will discuss how to extend this solution to a uniform family, in which the instance of 3-SAT($n$) to be solved is given as input to an EN P system $\Pi_{\text{3-SAT}}(n)$.

Let us start by showing how an EN P system can evaluate an instance of 3-SAT($n$) for a given assignment. For the sake of clarity, we will work with the following example instance of 3-SAT(4), defined over the set of variables $\{x_1, x_2, x_3, x_4\}$:

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4).$$

Notice that this instance is satisfiable: for example, the assignment $x_1 = x_2 = x_3 = \mathit{true}$ and $x_4 = \mathit{false}$ proves it.

To build this EN P system we simulate the following RAM, which uses registers $r_0, r_1, \ldots, r_9$, as follows:

- $r_0$ will contain the result of the computation: 1 if $\phi$ is satisfied by the given assignment, 0 otherwise;

- $r_1, r_2, r_3, r_4$ are assumed to be initialized with the truth values of variables $x_1, x_2, x_3, x_4$, respectively;
- $r_5$ is used to evaluate each clause. At the end of such evaluation, it contains 1 if the clause is *true* and 0 otherwise;
- $r_6, r_7, r_8$ are temporary registers used to evaluate a clause;
- $r_9$ is an auxiliary variable, used for branching in the program and to compute logical negations, whose value is always 1.

Assuming registers $r_1, r_2, r_3, r_4$ are initialized with the truth values of variables $x_1, x_2, x_3, x_4$, the RAM program illustrated in Algorithm 1 evaluates the above example instance $\phi$ of 3-SAT(4). Adapting this program to evaluate any fixed instance of 3-SAT($n$) for a given assignment is trivial: in this case, variable $r_0$ will contain the result of the evaluation, variables $r_1, \ldots, r_n$ will be initialized with the Boolean values assigned to $x_1, \ldots, x_n$, and variables $r_{n+1}, \ldots, r_{n+5}$ will be used to perform calculations, just like $r_5, \ldots, r_9$ for 3-SAT(4). Following the simulation of RAMs given in Theorem 2, we obtain the following all-parallel single-membrane EN P system $\Pi$ for the evaluation of the above example instance $\phi$ of 3-SAT(4):

$$\Pi = (1, H, \mu, (Var_1, Pr_1, Var_1(0))),$$

where:

- $H = \{1\}$
- $\mu = [\,]_1$
- $Var_1 = \{r_1, \ldots, r_9, z, p_0, \ldots, p_{36}\}$
- $Pr_1 = \{\ 0r_9 + 1 + z|_{p_0} \to 1|r_9,$
  $0r_6 + 2r_1 + z|_{p_1} \to 1|r_6 + 1|r_1,$
  $0r_7 + 2r_2 + z|_{p_2} \to 1|r_7 + 1|r_2,$
  $0r_8 + r_9 - r_3 + z|_{p_3} \to 1|r_8,$
  $r_9 + z|_{p_3} \to 1|r_9,$
  $r_3 + z|_{p_3} \to 1|r_3,$
  $r_6 - 1|_{p_4} \to 1|p_5,$
  $r_6 + 1|_{p_4} \to 1|p_6,$
  $0r_5 + 1 + z|_{p_5} \to 1|r_5,$
  $r_7 - 1|_{p_6} \to 1|p_7,$
  $r_7 + 1|_{p_6} \to 1|p_8,$
  $0r_5 + 1 + z|_{p_7} \to 1|r_5,$
  $r_8 - 1|_{p_8} \to 1|p_9,$
  $r_8 + 1|_{p_8} \to 1|p_{10},$
  $0r_5 + 1 + z|_{p_9} \to 1|r_5,$
  $r_5 - 1|_{p_{10}} \to 1|p_{11},$
  $r_5 + 1|_{p_{10}} \to 1|p_{35},$
  $0r_5 + 0 + z|_{p_{11}} \to 1|r_5,$
  $0r_6 + r_9 - r_1 + z|_{p_{12}} \to 1|r_6,$
  $r_9 + z|_{p_{12}} \to 1|r_9,$
  $r_1 + z|_{p_{12}} \to 1|r_1,$
  $0r_7 + 2r_3 + z|_{p_{13}} \to 1|r_7 + 1|r_3,$

$0r_8 + 2r_4 + z|_{p_{14}} \to 1|r_8 + 1|r_4,$
$r_6 - 1|_{p_{15}} \to 1|p_{16},$
$r_6 + 1|_{p_{15}} \to 1|p_{17},$
$0r_5 + 1 + z|_{p_{16}} \to 1|r_5,$
$r_7 - 1|_{p_{17}} \to 1|p_{18},$
$r_7 + 1|_{p_{17}} \to 1|p_{19},$
$0r_5 + 1 + z|_{p_{18}} \to 1|r_5,$
$r_8 - 1|_{p_{19}} \to 1|p_{20},$
$r_8 + 1|_{p_{19}} \to 1|p_{21},$
$0r_5 + 1 + z|_{p_{20}} \to 1|r_5,$
$r_5 - 1|_{p_{21}} \to 1|p_{22},$
$r_5 + 1|_{p_{21}} \to 1|p_{35},$
$0r_5 + 0 + z|_{p_{22}} \to 1|r_5,$
$0r_6 + r_9 - r_2 + z|_{p_{23}} \to 1|r_6,$
$r_9 + z|_{p_{23}} \to 1|r_9,$
$r_2 + z|_{p_{23}} \to 1|r_2,$
$0r_7 + 2r_3 + z|_{p_{24}} \to 1|r_7 + 1|r_3,$
$0r_8 + r_9 - r_4 + z|_{p_{25}} \to 1|r_8,$
$r_9 + z|_{p_{25}} \to 1|r_9,$
$r_4 + z|_{p_{25}} \to 1|r_4,$
$r_6 - 1|_{p_{26}} \to 1|p_{27},$
$r_6 + 1|_{p_{26}} \to 1|p_{28},$
$0r_5 + 1 + z|_{p_{27}} \to 1|r_5,$
$r_7 - 1|_{p_{28}} \to 1|p_{29},$
$r_7 + 1|_{p_{28}} \to 1|p_{30},$
$0r_5 + 1 + z|_{p_{29}} \to 1|r_5,$
$r_8 - 1|_{p_{30}} \to 1|p_{31},$
$r_8 + 1|_{p_{30}} \to 1|p_{32},$
$0r_5 + 1 + z|_{p_{31}} \to 1|r_5,$
$r_5 - 1|_{p_{32}} \to 1|p_{33},$
$r_5 + 1|_{p_{32}} \to 1|p_{35},$
$0r_0 + 1 + z|_{p_{33}} \to 1|r_0,$
$0r_0 + 0 + z|_{p_{35}} \to 1|r_0,$
$p_\ell \to 1|p_{\ell+1}$ for $\ell \in \{0, 1, \ldots, 33, 35\}$ }

– in $Var_1(0)$, all variables are initially set to 0, but variable $r_9$ which is initialized to 1, and variables $r_1, r_2, r_3, r_4$ which are initialized with the truth values assigned to the variables $x_1, x_2, x_3, x_4$ of $\phi$.

Notice that the translation from the RAM to the EN P system is mechanical. Albeit the EN P system is parallel the simulation of the RAM is performed one instruction at the time, due to the control imposed by variables $p_\ell$.

Let us now turn to the generation of all possible assignments for a given instance $\phi$ of 3-SAT($n$), and the parallel evaluation of $\phi$ for each of these assignments. Following the idea of [23], we now introduce (enzymatic) membrane

**Algorithm 1** Evaluation of the instance $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$ of 3-SAT(4)

**Require:** $r_1, r_2, r_3, r_4$ contain the truth values assigned to variables $x_1, x_2, x_3, x_4$
**Ensure:** $r_0 = 1$ if the instance is satisfied by the assignment, $r_0 = 0$ otherwise

0: $r_9 \leftarrow 1$ ▷ Initialization of the auxiliary register $r_9$
1: $r_6 \leftarrow r_1$ ▷ Evaluation of the first clause
2: $r_7 \leftarrow r_2$
3: $r_8 \leftarrow r_9 - r_3$
4: **if** $r_6 \neq 0$ **then** 5 **else** 6 ▷ If the first literal is *true* ...
5: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
6: **if** $r_7 \neq 0$ **then** 7 **else** 8 ▷ If the second literal is *true* ...
7: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
8: **if** $r_8 \neq 0$ **then** 9 **else** 10 ▷ If the third literal is *true* ...
9: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
10: **if** $r_5 \neq 0$ **then** 11 **else** 35 ▷ If the clause is *true* we continue, otherwise the entire formula is *false*
11: $r_5 \leftarrow 0$ ▷ Evaluation of the second clause
12: $r_6 \leftarrow r_9 - r_1$
13: $r_7 \leftarrow r_3$
14: $r_8 \leftarrow r_4$
15: **if** $r_6 \neq 0$ **then** 16 **else** 17 ▷ If the first literal is *true*...
16: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
17: **if** $r_7 \neq 0$ **then** 18 **else** 19 ▷ If the second literal is *true*...
18: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
19: **if** $r_8 \neq 0$ **then** 20 **else** 21 ▷ If the third literal is *true*...
20: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
21: **if** $r_5 \neq 0$ **then** 22 **else** 35 ▷ If the clause is *true* we continue, otherwise the entire formula is *false*
22: $r_5 \leftarrow 0$ ▷ Evaluation of the third clause
23: $r_6 \leftarrow r_9 - r_2$
24: $r_7 \leftarrow r_3$
25: $r_8 \leftarrow r_9 - r_4$
26: **if** $r_6 \neq 0$ **then** 27 **else** 28 ▷ If the first literal is *true*...
27: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
28: **if** $r_7 \neq 0$ **then** 29 **else** 30 ▷ If the second literal is *true*...
29: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
30: **if** $r_8 \neq 0$ **then** 31 **else** 32 ▷ If the third literal is *true*...
31: $r_5 \leftarrow 1$ ▷ ... then the clause is *true*
32: **if** $r_5 \neq 0$ **then** 33 **else** 35 ▷ If the clause is *true* then the entire formula is *true* otherwise the formula is *false*
33: $r_0 \leftarrow 1$
34: **accept**
35: $r_0 \leftarrow 0$
36: **reject**

division programs as follows:

$$F_{l,i}(x_{1,i},\ldots,x_{k_i,i})|_{e_{j,i}} \rightarrow [c_{l,1}|v_1 + c_{l,2}|v_2 + \cdots + c_{l,n_i}|v_{n_i}]_i + [c_{l,n_i+1}|v_1 + c_{l,n_i+2}|v_2 + \cdots + c_{l,2n_i}|v_{n_i}]_i,$$

In the above expression, a production function and a condition for an enzyme are the same as in a standard evolution program described above. After the application of the evolution program, a membrane labeled $i$ is divided into two membranes with the same label. Numerical variables and evolution programs are copied to the divided membranes, and numerical variables are changed according to the repartition protocols for each membrane. If the dividing membrane contains other membranes, we speak of *non-elementary* division; in this case, also these membranes are copied to the newly generated membranes. We assume that the skin membrane cannot be divided, i.e., an evolution program for membrane division for the outermost membrane cannot be defined on the EN P system. In addition, we also assume that numerical variables in inner or outer membranes cannot be changed in the evolution program for membrane division, for consistency of the EN P system.

Having extended the basic model of EN P systems by membrane division, we now follow the scheme defined by Gh. Păun in [19] to solve NP-complete problems: first, in $n$ computation steps the system builds $2^n$ elementary membranes in the skin membrane, each containing one of the possible assignments for the instance $\phi$ of 3-SAT($n$); then, in a number of steps that is proportional to the number of clauses present in the formula, each of these membranes evaluates $\phi$ for its assignment, and signals to the parent membrane whether the formula is satisfied. So doing, after a prescribed number of computation steps the value of a variable in the skin membrane will contain the number of satisfying assignments. Checking whether this value is positive answers the NP-complete problem, but since we know the number of satisfying assignments, we can also solve the #P-complete counting problem #3-SAT and the PP-complete decision problem THRESHOLD-3-SAT[2]. Through more complicated constructions we can use these P systems as oracles, reaching the solution of any problem in the PH polynomial hierarchy. However, we will not follow here this direction; we leave as future work explicit constructions that allow this to be done.

So, let us consider an EN P system $\Pi$ composed of a skin membrane, labelled 1, containing a single elementary membrane whose label is 2. Note that, since the described solution is semi-uniform, the P system $\Pi$ has no input and the instance $\phi$ of 3-SAT($n$) is encoded in the programs of membrane 2. The division of membrane 2 is driven by variables $d_{1,2},\ldots,d_{n,2}$, where in the initial configuration of the system $d_{1,2}$ is set to 1 and $d_{2,2},\ldots,d_{n,2}$ are set to 0. The division programs are:

$$z + 2|_{d_{i,2}} \rightarrow [1|x_{i,2} + 1|d_{i+1,2}]_2 + [1|d_{i+1,2} + 1|g]_2, \tag{6}$$

for $1 \leq i < n$, where $z$ is a variable that always equals 0, as above, and $g$ (which stands for *garbage*) is an auxiliary variable that receives excess values. When the

[2] This problem asks, given a 3-SAT($n$) formula $\phi$ and a positive integer $k < 2^n$, if there exist more than $k$ distinct truth assignments that satisfy $\phi$.

program in equation (6) is executed, membrane 2 generates two copies of itself. In the first copy the value of $x_{i,2}$ is 1, while in the second copy the value is 0. All other variables and programs originally contained in membrane 2 are copied as is into the two new copies of the membrane. The membrane division programs of equation (6) are executed in sequence, driven by variables $d_{1,2}, \ldots, d_{n,2}$. As a result, after $n$ computation steps $2^n$ copies of membrane 2 are created, each containing one of the possible assignments to the variables $x_{1,2}, \ldots, x_{n,2}$. The next (and last) membrane division program triggers the assignment check, setting the value of variable $p_{0,2}$ to 1:

$$z + 2|_{d_{n,2}} \to [1|x_{n,2} + 1|p_{0,2}]_2 + [1|p_{0,2} + 1|g]_2.$$

The assignment check then proceeds as indicated above, according to the scheme illustrated in Algorithm 1. When the verification is complete, if $r_0 = 1$ (corresponding to $\phi$ being satisfied by the assignment, and the execution of the **accept** statement), 1 is added to the variable *accept* in the skin:

$$z + 1|_{p_\ell} \to 1|accept \qquad \text{for instruction } \ell : \textbf{accept} \text{ of the RAM.}$$

Similarly, if $r_0 = 0$, corresponding to the fact that $\phi$ is not satisfied and thus the **reject** statement of the RAM is executed, 1 is added to the variable *reject* in the skin membrane:

$$z + 1|_{p_\ell} \to 1|reject \qquad \text{for instruction } \ell : \textbf{reject} \text{ of the RAM.}$$

Execution in each copy of membrane 2 terminates simultaneously, and therefore terminates in the entire P system, since there is no program in the skin membrane. It is easily seen that the P system operates in the all-parallel mode.

## 4 A Uniform Solution to 3-SAT by EN P Systems with Elementary Membrane Division

We now describe a uniform family $\mathbf{\Pi} = \{\Pi_{\text{3-SAT}}(n)\}_{n\in\mathbb{N}}$ of EN P systems such that each system $\Pi_{\text{3-SAT}}(n)$ can solve in polynomial time any instance of 3-SAT$(n)$.

To do this, we first need to find an appropriate encoding for the formulas of 3-SAT$(n)$. As recalled above, an instance of 3-SAT$(n)$ is a CNF Boolean formula $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$, where each clause $C_j = X_{j_1} \vee X_{j_2} \vee X_{j_3}$ is the disjunction of exactly three literals $X_{j_1}, X_{j_2}, X_{j_3}$ defined over the set of variables $Var(\phi) = \{x_1, \ldots, x_n\}$. In what follows, we will call 3-clauses this type of clause and, more generally, $k$-clauses those clauses that contain exactly $k$ literals. As said above, we can avoid the clauses in which the same literal is repeated or both the literals $x_i$ and $\neg x_i$, for any $1 \leq i \leq n$, occur. As a consequence, each clause of a 3-SAT instance can be obtained by considering a subset of $\{x_1, \ldots, x_n\}$ containing exactly three variables, each being subsequently negated or not. The number of all such possible subsets is $\binom{n}{3}$. Since the number of ways in which the

variables in a subset can be negated or not is eight, the total number of possible clauses in a 3-SAT($n$) formula is $8 \cdot \binom{n}{3}$, proportional to $n^3$, and thus polynomial in $n$. This means that a formula $\phi$ of 3-SAT($n$) can be represented as a binary string of length $m = 8 \cdot \binom{n}{3}$, where a bit equal to 1 means that the corresponding clause is part of the formula while a 0 means that it does not belong. This bit string will be the input given to the P system $\Pi_{\text{3-SAT}}(n)$ to specify the instance of 3-SAT($n$) to solve.

Let us now describe a specific encoding that allows us to compute the index $j \in \{0, 1, \ldots, m-1\}$ corresponding to a given 3-clause over $n$ variables and, vice versa, to compute the 3-clause corresponding to a given index. The proposed encoding will have the following property: the bit string representing all possible clauses of 3-SAT($n-1$) will be a *prefix* of the bit string representing the clauses of 3-SAT($n$). In other words, a formula of 3-SAT($n-1$) can be viewed as a formula of 3-SAT($n$) in which the variable $x_n$ does not appear. This means that the P system $\Pi_{\text{3-SAT}}(n)$ that solves all possible instances of 3-SAT($n$) is also able to solve all possible instances of 3-SAT($l$) for every $l \leq n$.

The bit string of length $m = 8 \cdot \binom{n}{3}$ that represents all possible clauses (and thus any possible instance) of 3-SAT($n$) is composed of $\binom{n}{3}$ *slots*, each of 8 bits. Each slot corresponds to a possible subset of three variables taken from $n$ variables. Denoted by $x_a, x_b, x_c$ the variables in a slot, the 8 bits of the slot correspond to the following clauses, read row-wise from left to right:

$$x_a \vee x_b \vee x_c \qquad x_a \vee x_b \vee \neg x_c \qquad x_a \vee \neg x_b \vee x_c \qquad x_a \vee \neg x_b \vee \neg x_c$$
$$\neg x_a \vee x_b \vee x_c \qquad \neg x_a \vee x_b \vee \neg x_c \qquad \neg x_a \vee \neg x_b \vee x_c \qquad \neg x_a \vee \neg x_b \vee \neg x_c$$

which can be easily seen to correspond to the 3-bit patterns $000, 001, \ldots, 111$, where a 1 indicates that the corresponding variable is negated.

Given a 3-clause defined over the variables $x_1, \ldots, x_n$, to determine the corresponding index $j \in \{0, 1, \ldots, m-1\}$ in the bit string that represents all possible clauses, we proceed as follows. We first determine the slot containing the three variables that compose the clause. Denoted by $s$ the slot number, we initially assume that $s$ is in the range $[1, \binom{n}{3}]$. Of the total $\binom{n}{3}$ slots, the first $\binom{n-1}{3}$ do not contain $x_n$ and the remaining $\binom{n-1}{2}$ do. Thus, if the given clause does not contain $x_n$ we recursively proceed with the first $\binom{n-1}{3}$ slots, assuming that $s$ is in the range $[1, \binom{n-1}{3}]$. Otherwise – that is, if the clause contains $x_n$ – we recursively proceed with the last $\binom{n-1}{2}$ slots, assuming that $s$ is in the range $[\binom{n}{3} - \binom{n-1}{2} + 1, \binom{n}{3}]$. The procedure ends when the lower and upper bound of the range coincide. Obtained in this way the slot number, we decrement it by 1, so that it is in the range $[0, \binom{n}{3} - 1]$ instead of $[1, \binom{n}{3}]$, we multiply it by 8 (the number of clauses contained in each slot), and we finally add the number in the range $[0, 7]$ that represents the negations in the clause.

To better fix the above procedure, let us consider as an example the clause $x_1 \vee \neg x_3 \vee x_5$, over $n = 6$ variables. Initially, the slot number $s$ is assumed to be in the range $[1, \binom{6}{3}] = [1, 20]$. Since the given clause does not contain $x_6$, $s$ must

**Algorithm 2** Calculating the index $j$ corresponding to a given 3-clause

**Require:** A 3-clause defined over $n$ Boolean variables, specified as follows:
- $v_1, v_2, v_3 \in \{1, 2, \ldots, n\}$: indicate the three variables that occur in the clause. We assume $1 \leq v_1 < v_2 < v_3 \leq n$;
- $neg_1, neg_2, neg_3 \in \{true, false\}$: indicate, for each of the three variables $v_1, v_2, v_3$, whether they occur negated or not in the clause.

**Ensure:** The index $j \in \{0, 1, \ldots, 8 \cdot \binom{n}{3} - 1\}$ associated with the input clause in the bit string that represents all possible 3-clauses of $n$ variables.

$v := n$ ▷ Variables to consider (can be put to $v := v_3$)
$k := 3$ ▷ Variables to be found
$l := 1$ ▷ The range for the slot number is $[l, u]$
$u := \binom{v}{k}$
**while** $l < u$ **do**
  **if** $v = v_3$ or $v = v_2$ or $v = v_1$ **then** ▷ $v$ is in the clause
    $l := u - \binom{v-1}{k-1} + 1$
    $k := k - 1$
  **else** ▷ $v$ is not in the clause
    $u := l + \binom{v-1}{k} - 1$
  **end if**
  $v := v - 1$
**end while**
$s := l$ ▷ The slot number, starting from 1
$j := (s - 1) \cdot 8$ ▷ Index at the beginning of the slot, starting from 0
**if** $neg_1$ **then** $j := j + 4$ ▷ Apply negations
**if** $neg_2$ **then** $j := j + 2$
**if** $neg_3$ **then** $j := j + 1$
**return** $j$

be among the first $\binom{5}{3} = 10$ slots, so we update the range to $[1, 10]$[3]. Since the clause contains $x_5$, we are in the last $\binom{4}{2} = 6$ slots, so we update the range to $[5, 10]$. Now the clause does not contain $x_4$, so we are in the first $\binom{3}{2} = 3$ slots, that is, in the range $[5, 7]$. The clause contains $x_3$, so the slot number is in the last $\binom{2}{1} = 2$ slots and we update the range to $[6, 7]$. The clause does not contain $x_2$, so we are in the first $\binom{1}{1} = 1$ slots and the range becomes $[6, 6]$. Since the lower and upper bounds in the range coincide, we can conclude that $s = 6$. If, despite this, we continue to analyze the clause, since it contains the variable $x_1$ we would find that the slot number $s$ is among the first $\binom{0}{0} = 1$ slots, and therefore in the range $[6, 6]$. The resulting index of the clause is finally obtained as $j = (6 - 1) \cdot 8 + 2 = 42$. The general procedure is illustrated in Algorithm 2.

On the other hand, given an index $j \in \{0, 1, \ldots, 8 \cdot \binom{n}{3} - 1\}$, to find the corresponding 3-clause we proceed as follows. We assume to know the number $n$

[3] Note that, since the indexes of clauses over variables $x_1, \ldots, x_5$ coincide with the indexes of the same clauses formally defined over the variables $x_1, \ldots, x_6$, we could have started by immediately considering the interval $[1, 10]$, since 10 is the number of 3-clauses that can be defined on five variables.

of variables upon which the instance $\phi$ is built, and of which the clause is part. On the contrary, if $n$ is unknown, it can be easily determined as the smallest positive integer such that $j > 8 \cdot \binom{n-1}{3}$. Then we determine the slot number in which $j$ resides: we first compute $j \mod 8$ to obtain the number corresponding to the negations of the variables in the clause; then, $s = (j - (j \mod 8)) \div 8 + 1$ gives the slot number in the range $[1, \binom{n}{3}]$. As before, of the $\binom{n}{3}$ total slots, the first $\binom{n-1}{3}$ do not contain $x_n$ and the remaining $\binom{n-1}{2}$ do; hence, the sought-after clause will contain $x_n$ if $s > \binom{n-1}{3}$ and will not contain it otherwise. Again, it is appropriate to maintain a range of slot numbers that narrows as the calculation progresses. Starting with the initial range $[1, \binom{n}{3}]$, we restrict it to $[\binom{n}{3} - \binom{n-1}{2} + 1, \binom{n}{3}]$ in the former case (the clause contains $x_n$) or to $[1, \binom{n-1}{3}]$ in the latter case (the clause does not contain $x_n$). We then proceed recursively on the new range, determining whether the clause contains $x_{n-1}$ (or not), then whether it contains $x_{n-2}$, and so on. The procedure ends when we have found all the three variables in the slot. The clause is finally obtained by applying the negations to the variables, as previously found.

As an example, we consider once again $j = 42$ for a formula of $n = 6$ variables. The slot number is given by $s = (j - (j \mod 8)) \div 8 + 1 = (42 - 2) \div 8 + 1 = 6$, and the initial range is $[1, \binom{6}{3}] = [1, 20]$. The pattern of negations of the three variables in the clause is $2 = 010$, that is, only the second variable is negated. Since $s \leq \binom{5}{3} = 10$, the clause does not contain $x_6$ and we update the range to $[1, 10]$. Now, since $s > \binom{4}{3} = 4$, the clause contains $x_5$ and we update the range to the last $\binom{5}{3} - \binom{4}{3} = 6$ slots, that is, $[5, 10]$. Having determined one of the variables that form the clause, we now have to look for two variables among the set of $\binom{4}{2} = 6$ slots that contain all possible 2-clauses of 4 variables. Which is composed of $\binom{3}{2} = 3$ slots (having numbers $5, 6, 7$) that do not contain $x_4$, and $6 - 3 = 3$ slots (having numbers $8, 9, 10$) that contain $x_4$. Since $s = 6$, the sought-after slot is in the first group, and thus the clause does not contain $x_4$. It now remains to find the slot among all possible slots of 2-clauses over 3 variables, which are $\binom{3}{2} = 3$, of which $\binom{2}{2} = 1$ (numbered 5) do not contain $x_3$ and the other $3 - 1 = 2$ (numbered $6, 7$) contain $x_3$. Since $s = 6$ is in the second group, our clause contains $x_3$ and we continue with the range $[6, 7]$. We now have to look for a 1-clause defined over a set of 2 variables. The total number of such clauses is $\binom{2}{1} = 2$, of which $\binom{1}{1}$ (numbered 6) does not contain $x_2$ and the other $2 - 1 = 1$ (numbered 7) contains $x_2$. Since $s = 6$ is in the first group, the clause does not contain $x_2$. We now must look for the slot number in the set of all possible 1-clauses of 1 variable, which contains $\binom{1}{1} = 1$ slot. Since the only possibility to compose the 1-clause is using the variable, which is $x_1$, we conclude that our clause contains $x_1$. We finally apply the negation pattern 010 to the three variables to obtain the clause $x_1 \vee \neg x_3 \vee x_5$. The general procedure is illustrated in Algorithm 3.

We are now ready to describe the uniform family $\mathbf{\Pi} = \{\Pi_{3\text{-SAT}}(n)\}_{n \in \mathbb{N}}$ of EN P systems in which each system $\Pi_{3\text{-SAT}}(n)$ solves in polynomial time any instance of 3-SAT($n$). Due to the complexity of the system $\Pi_{3\text{-SAT}}(n)$, we will just sketch its structure and operation, with the aim of convincing the reader

**Algorithm 3** Calculating the 3-clause corresponding to a given index $j$

**Require:** The index $j \in \{0, 1, \ldots, 8 \cdot \binom{n}{3} - 1\}$ associated with the sought-after clause in the bit string that represents all possible 3-clauses of $n$ variables

**Ensure:** The 3-clause corresponding to the index $j$ given as input, specified as follows:
- $v_1, v_2, v_3 \in \{1, 2, \ldots, n\}$: indicate the three variables that occur in the clause, with $1 \leq v_1 < v_2 < v_3 \leq n$;
- $neg_1, neg_2, neg_3 \in \{true, false\}$: indicate, for each of the three variables $v_1, v_2, v_3$, whether they occur negated or not in the clause.

$negs := j \mod 8$ ▷ Compute negations
**if** $negs \geq 4$ **then**
  $neg_1 := true$
  $negs := negs - 4$
**else**
  $neg_1 := false$
**end if**
**if** $negs \geq 2$ **then**
  $neg_2 := true$
  $negs := negs - 2$
**else**
  $neg_2 := false$
**end if**
**if** $negs = 1$ **then**
  $neg_3 := true$
**else**
  $neg_3 := false$
**end if**
$s := (j - (j \mod 8)) \div 8 + 1$ ▷ The slot number, starting from 1
$v := n$ ▷ Variables to consider
$k := 3$ ▷ Variables to be found
$l := 1$ ▷ The range for the slot number is $[l, u]$
$u := \binom{v}{k}$
**while** $k > 0$ **do**
  **if** $s > l + \binom{v-1}{k} - 1$ **then** ▷ $v$ is in the clause
    **if** $k = 3$ **then** $v_3 := v$
    **if** $k = 2$ **then** $v_2 := v$
    **if** $k = 1$ **then** $v_1 := v$
    $l := u - \binom{v-1}{k-1} + 1$
    $k := k - 1$
  **else** ▷ $v$ is not in the clause
    $u := l + \binom{v-1}{k} - 1$
  **end if**
  $v := v - 1$
**end while**
**return** $v_1$, $v_2$, $v_3$, $neg_1$, $neg_2$, $neg_3$

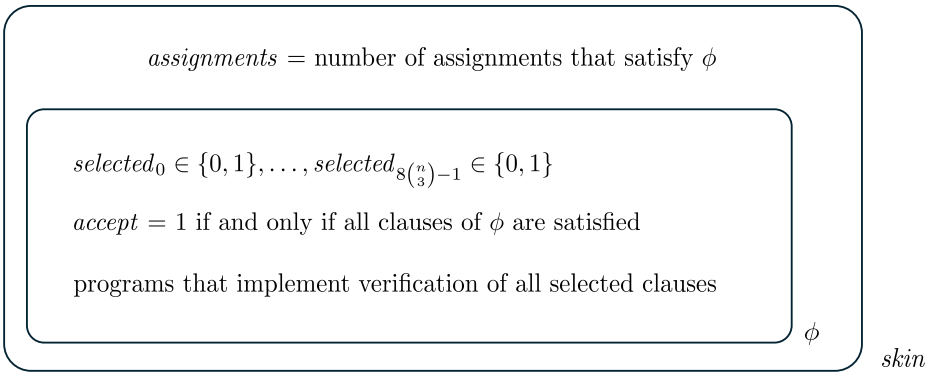



**Fig. 1.** Structure of the EN P system $\Pi_{3\text{-SAT}}(n)$ that solves all possible instances of 3-SAT$(n)$. During the computation, membrane $\phi$ is divided to generate $2^n$ copies, one for each possible assignment of variables $x_1, \ldots, x_n$. In this way, in each copy of membrane $\phi$ the clauses which have been selected as part of formula $\phi$ are evaluated on the corresponding assignment. The skin membrane receives from the $2^n$ copies of membrane $\phi$ the number of assignments that satisfy $\phi$, allowing it to simultaneously answer both the 3-SAT and the #3-SAT problems.

that such an EN P system can be built in polynomial time by a deterministic Turing machine, and that it actually solves any instance of 3-SAT$(n)$.

The structure of $\Pi_{3\text{-SAT}}(n)$ is illustrated in Figure 1. It is composed of a skin membrane, that contains a membrane with label $\phi$, whose structure is similar to the EN P system of Section 3 that checks whether a fixed 3-clause is satisfied by a given assignment: it contains a variable (named *accept*) which is set during the computation if and only if the assignment satisfies $\phi$, and the variables $selected_0, \ldots, selected_{8\binom{n}{3}-1}$ which are set to 1 in the initial configuration of the system if and only if the corresponding clause is part of the instance $\phi$ of 3-SAT$(n)$; the collection of these variables indeed constitutes the input to the P system $\Pi_{3\text{-SAT}}(n)$. Moreover, membrane $\phi$ contains the rules that allow it to evaluate the specified instance $\phi$ of 3-SAT$(n)$ on a given assignment to the variables $x_1, \ldots, x_n$, by evaluating in sequence all selected clauses. These rules are obtained from Algorithm 4, by first writing it as a RAM program[4] and then applying Theorem 2. All these transformations can be computed by a deterministic Turing machine in polynomial time while building the EN P system $\Pi_{3\text{-SAT}}(n)$. In fact, starting from the indexes $j \in \{0, 1, \ldots, 8 \cdot \binom{n}{3} - 1\}$ of the bit string that represents all possible 3-clauses, using Algorithm 3 the Turing machine can obtain each clause, then it can write a RAM program that evaluates it (similar to what we did in Algorithm 1), and finally using the transformation described in Theorem 2 it can obtain the actual rules.

[4] Note that the translation to a RAM program is very easy; here we preferred to write the algorithm in the form of pseudo-code for greater clarity.

**Algorithm 4** Evaluation of formula $\phi$ for a given assignment to the variables $x_1, \ldots, x_n$

**Require:** A CNF formula $\phi \in$ 3-SAT$(n)$, and an assignment to the variables $x_1, \ldots, x_n$ (where $1 = \mathit{true}$ and $0 = \mathit{false}$).

**Ensure:** A binary (0/1) value, stored in variable *accept*, which is 1 if and only if the specified assignment satisfies $\phi$.

clause$_0$: **if** $\mathit{selected}_0$ **then**
▷ Evaluate clause 0
**if** the clause is **not** satisfied **then goto** reject
**end if**
clause$_1$: **if** $\mathit{selected}_1$ **then**
▷ Evaluate clause 1
**if** the clause is **not** satisfied **then goto** reject
**end if**
⋮ ⋮
clause$_{8\binom{n}{3}-1}$: **if** $\mathit{selected}_{8\binom{n}{3}-1}$ **then**
▷ Evaluate clause $8\binom{n}{3} - 1$
**if** the clause is **not** satisfied **then goto** reject
**end if**
accept: $\mathit{accept} = 1$
**return**
reject: $\mathit{accept} = 0$
**return**

At the beginning of the computation, membrane $\phi$ divides itself to make $2^n$ copies in $n$ time steps, using division programs similar to those in equation (6):

$$z + 2|_{d_{i,\phi}} \to [1|x_{i,\phi} + 1|d_{i+1,\phi}]_\phi + [1|d_{i+1,\phi} + 1|g]_\phi,$$

for $1 \leq i < n$, where as usual $z$ is a variable that always equals 0 and $g$ is a "garbage" variable that receives excess values. As above, in the initial configuration of the system $d_{1,\phi}$ is set to 1 and $d_{2,\phi}, \ldots, d_{n,\phi}$ are set to 0. Each copy of membrane $\phi$ is thus associated with one possible assignment to the variables $x_1, \ldots, x_n$. The last membrane division program triggers the assignment check, by setting the value of variable $p_{0,\phi}$ to 1:

$$z + 2|_{d_{n,\phi}} \to [1|x_{n,\phi} + 1|p_{0,\phi}]_\phi + [1|p_{0,\phi} + 1|g]_\phi.$$

The assignment check then proceeds as described in Algorithm 4. When the check is complete, if the formula is satisfied by the assignment (corresponding to the execution of the **accept** statement), 1 is added to the variable *assignments* in membrane $\phi$:

$$z + 1|_{p_\ell} \to 1|\mathit{assignments}_{\mathit{skin}} \qquad \text{for instruction } \ell : \textbf{accept} \text{ of the RAM.}$$

Execution in each copy of membrane $\phi$ terminates simultaneously, so the variable *assignments* in the skin membrane receives at the same time the number

of assignments that satisfy $\phi$. So, the P system is not only able to solve the NP-complete decision problem 3-SAT but can also solve the #P-complete counting problem #3-SAT. It is also easily seen that the entire EN P system works in the all-parallel mode.

Note that passing the control of execution from one region (or sub-system) to another is made possible by the fact that the repartition protocol of a program can modify the variables that occur not only in the current region but also in neighboring (that is, parent or child) regions. This is a good thing, because it allows to design EN P systems in a modular way, increasing the expressiveness and usability of these models.

## 5 A Note on EN P Systems with Program Duplication

EN P systems operating in the all-parallel mode can be flattened, and thus reduced to single-membrane systems, as shown in Theorem 1. As mentioned above, flattening is achieved by adding a new index to each variable and each program in the P system, which refers to the membrane number (label) in the original P system. However, Theorem 1 holds for standard EN P systems, without membrane division programs. In this section, we argue whether flattening can also be applied to the extended version of EN P systems defined in Section 3. If that were the case, one could replace the generation of an exponential number of membranes with an equivalent generation of an exponential number of variables and programs.

Actually, our goal is a little different: we want to *avoid* generating an exponential number of membranes in the first place, and therefore avoid using programs for membrane division, at the cost of duplicating the programs (with the corresponding variables) present in the P system. Informally, this can be done as follows. Similar to what happens in flattening, an index is added to all the variables and programs that belong to the membrane that would be divided, which will assume unique values for each copy created. For example, the membrane division programs

$$z + 2|_{d_{i,2}} \to [1|x_{i,2} + 1|d_{i+1,2}]_2 + [1|d_{i+1,2} + 1|g]_2$$

would become:

$$z_k + 2|_{d_{i,2,k}} \to 1|x_{i,2,2k} + 1|d_{i+1,2,2k} + 1|d_{i+1,2,2k+1} + 1|g_{2k+1},$$

with $k$ starting from 1, with possibly some optimization, such as using a single variable $z$ instead of $z_k$ and a single variable $g$ instead of $g_{2k+1}$.

There are, however, two difficulties. First of all, if an exponential number of variables and programs are generated during computation, these must be present in the sets that constitute the definition of the P system. However, even in the case of defining a family $\mathbf{\Pi} = \{\Pi(n)\}_{n\in\mathbb{N}}$ of P systems that depend on the size $n$ of the instances of the problem to be solved, the number of variables and programs contained in each P system $\Pi(n)$ must be constructible in polynomial

time by a deterministic Turing machine, and therefore must be a polynomial number. A possible workaround could be to allow EN P systems to be defined via "schemas" of variables and programs, which can generate an exponential number of their instances during the computation. Should such definitions be prohibited or permitted? We leave this reflection to the reader, as the author has no strong opinion one way or the other. Note that the case considered here is specifically constructed to solve 3-SAT, and can probably be easily adapted to solve any NP-complete problem, using the technique introduced in [19]. We note, however, that if this type of definition were allowed, then the computational power of EN P systems that can use an exponential number of variables and/or programs would need to be determined. If their problem-solving ability were limited by PSPACE (the class of problems solvable in *polynomial time* by EN P systems that use the arithmetic operations $+$, $-$, $\times$, and $\div$), then the use of membrane division would only increase the expressiveness of these systems, making them easier to use. If, however, the use of an exponential number of variables and/or programs were to allow them to go beyond PSPACE (perhaps characterizing EXPSPACE?), then these systems should be used with greater caution. Establishing the exact computational power of EN P systems that can use an exponential number of variables and/or programs remains an interesting open question. Note also that in this paper we have just used division for *elementary* membranes, that is, membranes not containing any other membrane. Does the computing power increase when allowing division for non-elementary membranes? This is another open question.

In any case, a second obstacle seems to be difficult or even impossible to overcome. To simulate the division of a membrane it is not enough to duplicate a single program and all the variables involved in it: instead, *all* the programs and variables within the (simulated) membrane which is going to be divided must be duplicated. In the flattened version of the P system, this means arbitrarily selecting subsets of programs and variables to duplicate. The same problem arises when we aim to flatten a standard EN P system that works in the sequential mode, as highlighted in [10]. Maybe this problem can be solved by introducing a new control mechanism, perhaps some form of aggregation in classes for programs (possibly through aggregated and coordinated enzymes?), but it remains an interesting direction for future research.

## 6 Conclusions and Directions for Future Work

In this paper, we investigated enzymatic numerical (EN) P systems with membrane division, and we used them to solve the NP-complete problem 3-SAT in polynomial time. In fact, although it is known that EN P systems using the arithmetic operations $+$, $-$, $\times$, $\div$ are able to solve all PSPACE problems in polynomial time [9], the construction to do so is rather abstract, and a direct construction to solve even NP-complete problems by these systems is missing in the literature (with the notable exception of [23], for solving the SUBSET SUM problem). Thus, in Section 3 we first presented a semi-uniform solution, where

each P system is able to solve a unique instance $\phi$ of 3-SAT($n$), encoded in its programs. Then, in Section 4, we sketched the structure of a uniform solution, where each P system can solve all the instances of 3-SAT($n$) defined on $n$ Boolean variables.

As observed in Section 3, the proposed systems actually compute the number of assignments that satisfy the 3-SAT($n$) formula. While checking whether this value is positive answers the NP-complete problem, with trivial modifications to the programs of the systems we can also solve the #P-complete counting problem #3-SAT and the PP-complete decision problem THRESHOLD-3-SAT[5]. Through more complicated constructions we could use these P systems as oracles, reaching the solution of any problem in the PH polynomial hierarchy; we leave as future work explicit constructions that allow this to be done.

All the proposed P systems work in the all-parallel mode, and their extension to solve $k$-SAT for any $k \geq 3$ is trivial. Since this kind of systems can be flattened, that is, reduced to a single membrane, in Section 5 we discussed the possibility of solving NP-complete problems by single-membrane EN P systems with program (and variable) duplication that mimick the operation of EN P systems with membrane division. Some technical problems make this difficult, and it is not clear whether they can be overcome. In author's opinion, this is an interesting direction for future work.

On the other hand, since both membrane division and program and variable duplication create an exponential-size workspace in polynomial time, it is important to determine the actual computational power of these systems. As pointed out in Section 5, if their power is capped by PSPACE, then membrane division and/or program and variable duplication just become useful tools to increase the expressiveness and, therefore, the practical usability of EN P systems in problem solving. But if their computational power goes beyond PSPACE, then greater attention is needed in using them, and determining their precise computational power becomes another interesting open problem.

Finally, it would be interesting to explicitly build in detail the EN P systems proposed in this paper, especially the one for the uniform solution of 3-SAT as well as the variant that solves the #P-complete counting problem #3-SAT, so that they can be run in a simulator such as PeP [5], GPUPeP [22], or SNUPS [1]. Note that this explicit construction would also involve simulating the deterministic Turing machine that uses Algorithm 3 and Theorem 2 to build the EN P systems.

**Disclosure of Interests.** The author has no competing interests to declare that are relevant to the content of this article.

## References

1. Arsene, O., Buiu, C., Popescu, N.: SNUPS – A simulator for Numerical Membrane Computing. International Journal of Innovative Computing, Information and Control (IJICIC) **7**(6), 3509–3522 (2011)

[5] This problem asks, given a 3-SAT($n$) formula $\phi$ and a positive integer $k < 2^n$, if there exist more than $k$ distinct truth assignments that satisfy $\phi$.

2. Bertoni, A., Mauri, G., Sabadini, N.: A characterization of the class of functions computable in polynomial time on random access machines. In: STOC '81 Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing. pp. 168–176 (1981)
3. Buiu, C., Vasile, C.I., Arsene, O.: Development of membrane controllers for mobile robots. Information Sciences **187**, 33–51 (2012)
4. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. Journal of Computer and System Sciences **7**, 354–375 (1973)
5. Florea, A.G., Buiu, C.: PeP - An open-source software simulator of Numerical P systems and Numerical P systems with Enzymes. `https://github.com/andrei91ro/pep` (2017)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco (1979)
7. Hartmanis, J., Simon, J.: On the power of multiplication in random access machines. In: IEEE Conference Record of 15th Annual Symposium on Switching and Automata Theory. pp. 13–23 (1974)
8. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae **71**, 279–308 (2006)
9. Leporati, A., Mauri, G., Porreca, A.E., Zandron, C.: Enzymatic numerical P systems using elementary arithmetic operations. In: Revised Selected Papers of the 14th International Conference on Membrane Computing – LNCS Volume 8340. pp. 249–264. CMC 2013, Springer-Verlag, Berlin, Heidelberg (2013)
10. Leporati, A., Porreca, A.E., Zandron, C., Mauri, G.: Improved universality results for parallel enzymatic numerical P systems. International Journal of Unconventional Computing **9**(5-6), 385–404 (2013)
11. Liu, J., Wang, L., Zhang, G., Verlan, S., Zhu, M.: Universal enzymatic numerical P systems with small number of enzymatic rules. Theoretical Computer Science **1004**, 114630 (2024)
12. Martín-Vide, C., Păun, G., Pazos, J., Rodríguez-Patón, A.: Tissue P systems. Theoretical Computer Science **296**(2), 295–326 (2003)
13. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1993)
14. Păun, A., Păun, G.: The power of communication: P systems with symport/antiport. New Generation Computing **20**(3), 295–305 (2002)
15. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000)
16. Pavel, A.B., Arsene, O., Buiu, C.: Enzymatic numerical P systems - A new class of membrane computing systems. In: Li et al., K. (ed.) Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on. pp. 1331–1336 (2010)
17. Pavel, A.B., Buiu, C.: Using enzymatic numerical P systems for modeling mobile robot controllers. Natural Computing **11**(3), 387–393 (2012)
18. Pavel, A.B., Vasile, C.I., Dumitrache, I.: Robot localization implemented with enzymatic numerical P systems. In: Prescott et al., T.J. (ed.) Biomimetic and Biohybrid Systems, LNCS, vol. 7375, pp. 204–215. Springer Berlin Heidelberg (2012)
19. Păun, G.: P systems with active membranes: attacking NP-complete problems. Journal of Automata, Languages and Combinatorics **6**(1), 75–90 (2001)
20. Păun, G., Păun, R.: Membrane computing and economics: Numerical P systems. Fundamenta Informaticae **73**(1–2), 213–227 (July 2006)
21. Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, New York, NY, USA (2010)

22. Raghavan, S., Rai, S.S., Rohit, M.P., Chandrasekaran, K.: GPUPeP: Parallel Enzymatic Numerical P System simulator with a Python-based interface. BioSystems **196**, 104186 (2020)
23. Shiiba, T., Fujiwara, A.: Solving Subset Sum problem using EN P system with active membranes. In: 2016 Joint 8th International Conference on Soft Computing and Intelligent Systems (SCIS) and 17th International Symposium on Advanced Intelligent Systems (ISIS). pp. 886–891 (2016)
24. Vasile, C.I., Pavel, A.B., Dumitrache, I.: Universality of enzymatic numerical P systems. International Journal of Computer Mathematics **90**(4), 869–879 (2013)
25. Vasile, C.I., Pavel, A.B., Dumitrache, I., Păun, G.: On the power of enzymatic numerical P systems. Acta Informatica **49**(6), 395–412 (2012)

# Toward Partition-Aware Design of Spiking Neural P Systems: A Bulk Synchronous Parallel Cost Model and a Binary Multiplication Case Study

Job J. Lipat[1], Henry N. Adorna[1], Francis George C. Cabarle[1], Miguel Ángel Martínez-del-Amor[2], and David Orellana-Martín[2]

[1] Department of Computer Science, College of Engineering, University of the Philippines Diliman, 1101 Quezon City, Philippines
jjlipat@up.edu.ph, hnadorna@up.edu.ph, fccabarle@up.edu.ph

[2] Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Seville, Spain
mdelamor@us.es, dorellana@us.es

**Abstract.** Distributed execution of a spiking neural P (SN P) system assigns neurons to processing components, so spikes sent across component boundaries become messages. Standard correctness, size, and step measures do not reveal whether a construction admits a communication-efficient balanced placement. We address this gap by adding a static neuron-to-block map to the SN P model, showing that the distributed execution it induces has the same possible computations and outputs as the original SN P system, and specializing Bulk Synchronous Parallel (BSP) cost to local work, cross-block spike traffic, and synchronization. We then define a partition-aware criterion that compares structure-aware and structure-blind balanced layouts over a stated computation family. As a case study, we analyze the Zhang shift–add multiplier for $n$-bit unsigned operands. When both operands are all ones, every asymptotically balanced partition family into a fixed number $K \geq 2$ of blocks has cumulative BSP communication $H(P_n) = \Omega(n^2)$. The lower bound follows from linearly many replicated shift paths that remain active throughout the input window and meet at shared endpoints, showing that improved placement alone cannot eliminate the quadratic communication. Guided by this diagnosis, we outline a carry-save multiplier based on temporal bit reuse, bounded-fan-in reducers, and ordered lanes that expose a structure-aware layout. A preliminary analysis of the sketched topology and schedule suggests that consecutive lane bands can be balanced with cumulative communication $O(n \log n)$ for fixed $K$. A complete construction, correctness proof, and end-to-end BSP cost remain future work.

**Keywords:** Spiking neural P systems · membrane computing · graph partitioning · Bulk Synchronous Parallel · binary multiplication

## 1 Introduction

Spiking neural P (SN P) systems are synchronized directed graphs whose neurons exchange indistinguishable spikes; firing rules determine when spikes are emitted [6,8]. Constructions are usually judged by their function, neuron and rule counts, and global steps. But when scale or deployment calls for distributed execution, these measures are incomplete: systems with comparable step counts can incur different communication and local-work costs.

In that setting, a *block* is a group of neurons assigned to one processing component, and the block containing a neuron is its *owner*. A synapse is *internal* when its two endpoint neurons have the same owner. Otherwise it belongs to the *cut*, the set of synapses crossing between blocks, and every spike it carries must be communicated as a message. Partitioning spreads computation and memory across components. It helps only when the partition is *balanced*, meaning the blocks hold roughly equally many neurons, and when most active traffic stays inside them.

Matrix and graphics processing unit (GPU) simulators generally treat the synapse graph as a flat input [14,1]. Our recent simulator, built with the Message Passing Interface (MPI) and NVIDIA's CUDA platform, partitions that graph after construction; partition choice had little effect on its densely connected sorting topology [9]. This motivates two questions: how can placement and distributed cost be made explicit in an SN P system, and how can they guide future constructions?

We answer in three steps. First, a *partitioned SN P system* adds a fixed neuron-to-block map and defines a distributed execution with the same possible computations and outputs as the original SN P system (Section 3). Second, the Bulk Synchronous Parallel (BSP) model [12] provides a cost-accounting framework for this execution. Each block maps to a component, and each SN P step is one *superstep* comprising local work, cross-block messages, and the synchronization required by the global clock (Section 4). Third, for a stated computation family, we call a construction partition-aware when a balanced *structure-aware* layout, which may use the synapse topology when grouping neurons, saves a fixed positive fraction of total BSP cost compared with a *structure-blind* baseline, which assigns neurons without using that topology (Section 5).

Unsigned multiplication provides the test (Section 6). We use the Zhang shift–add multiplier as the reference construction. It replicates rows that delay an input and uses highly connected neurons [16]. For $n$-bit operands, we show that every balanced placement into a fixed number $K$ of blocks cuts linearly many active rows when both operands are all ones, which we call the *all-one input* (Section 7). The quadratic communication comes from many active paths that lead to the same neurons. Our closing carry-save multiplier sketch (Section 8) instead reuses stored bits in ordered lanes that a partition can follow and reduces three streams to two at each stage. A preliminary analysis in Section 9 suggests that its intended topology and schedule can support balanced lane bands with $O(n \log n)$ cumulative communication for fixed $K$. The complete construction and its correctness proof remain future work.

**Contributions.** This paper makes three contributions. First, it defines partitioned SN P systems, establishes placement invariance, and formulates a BSP simulation cost and a partition-aware comparison criterion. Second, it demonstrates that, on the all-one input and for fixed $K$, the cumulative BSP communication $H(P_n)$ of the Zhang shift–add multiplier is $\Omega(n^2)$ for every family of placements that stays balanced as $n$ grows. Third, it extracts design requirements from the structural source of this lower bound, applies them in a prospective carry-save multiplier, and derives a preliminary $O(n \log n)$ communication estimate for its intended lane-band placement.

## 2 Related Work: From SN P Systems to Distributed Execution

Distributing an SN P system across multiple processing components requires a decision about where each neuron resides. Synapses within one component can be simulated locally, whereas spikes crossing component boundaries must be exchanged as messages. Before formalizing this distinction, we trace how existing work moves from SN P semantics to executable representations and communication-aware placement, then to graph partitioning and BSP descriptions of placement and cost. We close by distinguishing the neuron-level placement studied here from SN dP systems, in which several SN P systems act as cooperating modules.

Ionescu, Păun, and Yokomori introduced the standard model of spiking neural P systems [6]; Leporati, Mauri, and Zandron survey its principal variants and complexity results [8]. The model specifies neurons, rules, synapses, and a global clock, which together determine the system's transitions. It abstracts from deployment: neurons have no processor owners, and the definition does not distinguish a local synapse from a remote one. Distributed placement is therefore an implementation decision, not part of the standard SN P description.

Simulation research moves from that description to an executable representation. Zeng et al. encode system states, called configurations, rule choices, and synapses using vectors and matrices, so a transition can be evaluated by linear-algebra operations [14]. Martínez-del-Amor et al. develop compressed sparse-matrix representations for fixed and changing SN P structures, reducing the matrix data that a simulator must store and process [1]. These approaches begin with a specified SN P system and derive the data structures used to execute it.

Distributed execution makes the placement decision unavoidable. In earlier work, we developed a hybrid MPI and CUDA simulator that partitions the synapse graph across processing components and represents remote neurons adjacent to crossing synapses by local copies [9]. On its natural-number sorting benchmark, contiguous identifier ranges, the community-based Louvain method, and Red–Blue breadth-first search produced similar overall performance. For most input distributions, execution times for the naive distributed implementation differed by less than 10% among the three strategies. The benchmark topology was dense and regularly ordered, so contiguous neuron identifiers al-

ready provided a useful placement and the more elaborate partitioners offered little consistent advantage. This result is limited to one topology: it does not establish that contiguous placement is generally best. Instead, it illustrates why the structure presented to the partitioner matters.

The same concern, communication-aware placement, appears in neighboring models. Fernandez-Musoles et al. use hypergraph partitioning and sparse communication for distributed spiking neural network simulation [4]; Zeng et al. combine area-based partitioning with topology-aware mapping for the same setting [15]; and Muniyandi and Maroosi cluster communicating membrane-computing work for GPU execution [11]. These studies optimize simulator or hardware mappings rather than define neuron ownership in an SN P system, but they show how communication cost can depend on placement.

Graph partitioning provides a natural way to reason about that structure. A balanced partition seeks a small edge cut subject to a balance constraint [2]. For an SN P simulation, however, edge cut is only a static proxy for communication. A crossing synapse produces a message when its source emits a spike, and the same synapse may be used repeatedly during a computation. Communication therefore depends on both placement and the sequence of emissions, not on the number of crossing synapses alone.

BSP provides the cost vocabulary used in Section 4. Its superstep cost separates the largest local workload, the largest send or receive volume, and synchronization overhead; the $h$-relation supplies the communication volume [12,10]. Cremonesi and Schürmann similarly analyze large-scale neural simulations under BSP and model spike exchange in greater detail [3].

One terminological boundary is important before introducing our definition. Ionescu et al. define spiking neural distributed P (SN dP) systems, in which several SN P systems act as cooperating modules, receive distributed parts of the input, and communicate through external synapses [5]. Our use of *partitioned* begins instead with one SN P system whose individual neurons are assigned to blocks for distributed execution. Thus, SN dP systems distribute input and cooperation across SN P modules, whereas a partitioned SN P system records the deployment of one computation graph.

Standard SN P semantics therefore fixes what the system computes, while simulation, mapping, graph partitioning, and BSP explain where it executes and what placement can cost. The next section connects these views by attaching neuron ownership to one SN P system and showing that the distributed execution this induces has the same possible computations and outputs as the original SN P system.

## 3 Partitioned SN P Systems

How can an SN P system run on several components and still compute the same thing? We answer with a static neuron-to-block map $P$ that turns block-crossing deliveries into messages, after recalling the transition semantics needed to define it.

### 3.1 The standard SN P model

**Definition 1 (SN P system [6]).** *An SN P system with $m \geq 1$ neurons is*

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, i_0),$$

*where $O = \{a\}$ is the spike alphabet, neuron $\sigma_i = (n_i, R_i)$ initially contains $n_i$ spikes and has rule set $R_i$, $syn \subseteq \{1, \ldots, m\}^2$ is the directed synapse relation without self-loops, and $i_0$ is the output neuron. A firing rule has form $E/a^c \to a; d$: $E$ is a regular expression describing the spike counts that enable the rule, $c \geq 1$ spikes are consumed, and one spike is emitted after delay $d \geq 0$. A forgetting rule $a^s \to \lambda$ is applicable only when the neuron contains exactly $s \geq 1$ spikes and removes them without emitting any; as in the standard definition, $a^s \notin L(E)$ for every firing rule $E/a^c \to a; d$ in the same neuron.*

We use two extensions of this rule syntax. The form $E/a^c \to a^p; d$ emits $p \geq 1$ spikes, following the extended variant surveyed by Leporati, Mauri, and Zandron [8]. The form $E/a^c \to \lambda$ consumes $c$ spikes without emitting any when the current spike count belongs to $L(E)$. The standard forgetting rule $a^s \to \lambda$ is therefore the special case $a^s/a^s \to \lambda$.

Under the standard transition semantics [6], a global clock drives rule application. At each step, every neuron that is not waiting through a delay (an *open* neuron) and has at least one applicable rule selects one; the choice is nondeterministic if several apply. A firing neuron is *closed*, or refractory, during its delay. Its eventual emission is directed to every neuron connected by an outgoing synapse: an open target incorporates the spikes into the next configuration, while a closed target discards them. A configuration records every neuron's spike count and remaining refractory time.

### 3.2 The partitioned SN P model

We now introduce the partitioned extension used in this paper.

**Definition 2 (Partitioned SN P system).** *A partitioned SN P system with $K$ blocks is*

$$\Pi_P = (O, \sigma_1, \ldots, \sigma_m, syn, P, i_0),$$

*where the tuple without $P$ is an SN P system and $P : \{1, \ldots, m\} \to \{1, \ldots, K\}$ is a surjective static partition map, so every block owns at least one neuron. Block $b$ is $V_b = \{i : P(i) = b\}$.*

Under $P$, a synapse $(i, j)$ is internal when $P(i) = P(j)$. The remaining synapses form the cut

$$cut(P) = \{(i, j) \in syn : P(i) \neq P(j)\}.$$

The map by itself is only a label on neurons. What it induces is a way of running the system, the *distributed execution* of $\Pi_P$: each block runs on its

own component. Suppose neuron $i$ emits $p$ spikes at step $t$. A delivery along an internal synapse stays inside one component. For every $(i, j) \in cut(P)$, the delivery instead becomes a message from $P(i)$ to $P(j)$ carrying the target $j$ and $p$ spikes. Components exchange these messages and synchronize before step $t+1$, so a message arrives at the step an internal delivery would, and a closed target likewise discards the spikes.

Let $p_i(t)$ be the nonnegative integer number of spikes emitted by neuron $i$ at step $t$, with $p_i(t) = 0$ when it emits nothing. The number of spikes sent across cut synapses at that step is

$$C_P(t) = \sum_{(i,j)\in cut(P)} p_i(t).$$

Thus the static cut records where communication can occur, whereas $C_P(t)$ records the cross-block traffic generated by a particular computation.

### 3.3 Preservation under distributed execution

**Proposition 1 (Placement invariance).** *Let $U(\Pi_P)$ be the underlying system obtained by forgetting $P$. The distributed execution of $\Pi_P$ and the standard semantics of $U(\Pi_P)$ have the same computations and the same output spike trains, meaning the same sequence of output-neuron emissions.*

*Proof.* The two systems have the same neurons, rules, and synapses, hence the same configurations. Fix a configuration and the same rule choices in both. Applicability depends only on spike counts and refractory times, so both emit the same $p_i(t)$, and deliveries on internal synapses coincide. For $(i, j) \in cut(P)$, the message carries target $j$ and $p_i(t)$ spikes, and the synchronization before step $t + 1$ delivers it when the standard semantics would; a closed $\sigma_j$ discards it in both. The successor configuration is therefore the same, and induction on $t$ gives the same computations and the same emissions of $\sigma_{i_0}$. □

**Corollary 1.** *All partitions of the same underlying system have the same possible computations and results. Under the same rule choices, they produce the same output in the same number of steps. They can differ only in the placement of work and communication.*

Running the blocks as communicating components therefore computes what the undivided system computes. This allows placements to be compared without conflating them with different computations. We next ask what distributed cost a placement induces.

## 4 BSP Cost of a Partitioned Computation

The SN P semantics determines the computation, while BSP provides a framework for accounting for its distributed cost. It separates the maximum local work and message volume at any component from the cost of synchronization.

The global SN P clock gives a natural boundary for each BSP superstep. We assign $V_b$ to component $b$. During one SN P step, each component performs local work, exchanges spikes on cut synapses as messages, and synchronizes before the next step. Thus, one SN P step is one superstep in this model. We use the following quantities at step $t$:

| Quantity | What it counts |
|---|---|
| $p_i(t)$ | Spikes emitted by neuron $i$, with $p_i(t) = 0$ when it emits nothing. |
| $\rho_b(t)$ | Rule applications by neurons in $V_b$. |
| $\delta_b(t)$ | Spike deliveries on internal synapses in $V_b$. |
| $\mathrm{sent}_b(t)$ | Spikes sent from block $b$ across cut synapses: $\sum_{(i,j)\in cut(P),\, P(i)=b} p_i(t)$. |
| $\mathrm{recv}_b(t)$ | Spikes received by block $b$ across cut synapses: $\sum_{(i,j)\in cut(P),\, P(j)=b} p_i(t)$. |

Our local-work model assumes a step-driven simulator. At every SN P step, component $b$ inspects all $|V_b|$ neurons it owns, whether or not they apply a rule. It also applies selected rules, handles internal spike deliveries, and processes received cross-block spikes at their target neurons. Message transmission between components is counted separately as communication.

**Definition 3 (Local work).** *Let $\kappa_n, \kappa_r, \kappa_i, \kappa_x \geq 0$ be the unit costs for inspecting one neuron, applying one rule, delivering one spike on an internal synapse, and processing one received cross-block spike at its target, respectively. Define*

$$w_b(t) = \kappa_n |V_b| + \kappa_r \rho_b(t) + \kappa_i \delta_b(t) + \kappa_x \,\mathrm{recv}_b(t), \qquad w(t) = \max_b w_b(t).$$

Following the BSP $h$-relation, which measures the largest send or receive volume at any component [12], define

$$h_b(t) = \max\{\mathrm{sent}_b(t), \mathrm{recv}_b(t)\}, \qquad h(t) = \max_b h_b(t).$$

Summed over all blocks, both the sent and received counts equal $C_P(t)$. Averaging therefore gives

$$\frac{C_P(t)}{K} \leq h(t) \leq C_P(t). \tag{1}$$

**Definition 4 (BSP simulation cost [10]).** *For a $T$-step computation, let $g, \ell \geq 0$. The parameter $g$ is the per-spike communication cost, and $\ell$ is the fixed synchronization cost of the barrier after each SN P step, independent of $h(t)$. Define*

$$\begin{aligned} \mathit{Cost}(\Pi_P; g, \ell) &= \sum_{t=1}^{T} \big(w(t) + g h(t) + \ell\big) \\ &= \sum_{t=1}^{T} \max_b w_b(t) + g \sum_{t=1}^{T} h(t) + \ell T. \end{aligned} \tag{2}$$

*Write*

$$W(P) = \sum_{t=1}^{T} w(t) = \sum_{t=1}^{T} \max_{b} w_b(t), \qquad H(P) = \sum_{t=1}^{T} h(t).$$

*Thus, $W(P)$ accumulates the local-work bottleneck, $gH(P)$ the communication cost, and $\ell T$ the cost of the $T$ synchronization barriers:*

$$Cost(\Pi_P; g, \ell) = W(P) + gH(P) + \ell T.$$

*For a fixed computation, $\ell T$ is the same for every placement, whereas $gH(P)$ depends on the cross-block spike traffic induced by $P$.*

**Proposition 2 (Partition-dependent terms).** *For maps $P$ and $P'$ with the same number of blocks on one underlying system, consider the same $T$-step computation, with the same input and rule choices, under both maps. Then*

$$Cost(\Pi_{P'}; g, \ell) - Cost(\Pi_P; g, \ell) = W(P') - W(P) + g\big(H(P') - H(P)\big).$$

*Proof.* By Proposition 1, the two placements have the same computation and $T$. Their $\ell T$ terms are therefore equal and cancel when the two costs in Eq. (2) are subtracted. □

The expression compares two placements, but not yet entire construction families. That requires an asymptotic design criterion.

## 5 Partition-Aware Design Criterion

When does topology permit increasingly cheaper placement as the input grows? Consider a family $\{\Pi_n\}_{n\geq 1}$ of SN P systems, where $n$ measures the size of the input handled by $\Pi_n$. We compare two placement families satisfying the balance condition below that differ in whether their maps may inspect the synapse topology.

Balanced graph partitioning conventionally limits block sizes while seeking a small edge cut [2]. For the family-level comparison in this paper, we use the following asymptotic balance condition.

**Definition 5 (Asymptotically balanced partition family).** *Let $N(n)$ be the total number of neurons in $\Pi_n$, and let $V_b$ be the neuron set assigned to block $b$. For a fixed number $K$ of blocks, a partition family is asymptotically balanced if, for some nonnegative allowed excess $\beta(n) = o(N(n))$, meaning $\beta(n)/N(n) \to 0$,*

$$|V_b| \leq \left\lceil \frac{N(n)}{K} \right\rceil + \beta(n) \qquad \textit{for every block } b. \tag{3}$$

Because the fixed $K$ blocks contain all neurons, this also implies $|V_b| = N(n)/K + o(N(n))$.

Balance is a requirement shared by the two placement classes; the information available to their maps distinguishes them. We make that distinction explicit with the following paper-specific terminology.

**Definition 6 (Structure-blind and structure-aware partition families).** *Fix $K$ and require both classes to be asymptotically balanced. A* structure-blind *family has maps that may use neuron identifiers, $N(n)$, and $K$, but may not inspect the synapse relation. Assigning contiguous identifier ranges or cycling through the blocks are examples. A* structure-aware *family may additionally inspect the synapse relation when assigning neurons to blocks. Thus,* structure *here means the directed synapse topology. Each comparison designates one particular structure-blind family $(P_n)$ as its baseline.*

Because traffic depends on spike activity, exact communication cost is tied to a computation. We therefore fix one computation of $\Pi_n$ for each $n$, including its input and any nondeterministic choices, and fix $g$ and $\ell$. For structure-aware maps $P'_n$ and the stated structure-blind baseline maps $P_n$, evaluate that same computation using Definition 4. Assuming the baseline cost is positive for all sufficiently large $n$, define the relative saving of $P'_n$ over $P_n$ by

$$R_n(P', P) = 1 - \frac{Cost(\Pi_{n,P'_n}; g, \ell)}{Cost(\Pi_{n,P_n}; g, \ell)}.$$

**Definition 7 (Partition-aware family).** *A construction family is partition-aware for $K$ blocks and the stated computation family if it admits a structure-aware family $(P'_n)$ and a constant $\varepsilon > 0$ such that, against the stated structure-blind baseline $(P_n)$,*

$$R_n(P', P) \geq \varepsilon$$

*for all sufficiently large $n$.*

Partition awareness is therefore relative to $K$, the stated computations, and the named baseline. Both placement families must remain balanced, and using the synapse topology must save a fixed positive fraction of the baseline's total BSP cost. Because a cut synapse communicates only when its source fires, the cost comparison depends on spike activity and local-work balance as well as the static cut.

With a known $T$-step bound, the rules and a candidate placement give a communication upper bound before execution. Let $\widehat{F}_i$ bound the total spikes emitted by neuron $i$ during those $T$ steps. One general choice is $T$ times the largest number of spikes emitted by any rule in neuron $i$, since a neuron applies at most one rule per step. Input and schedule analysis can tighten this bound by limiting which rules can apply and when their neurons are active. Equation (1) then gives

$$H(P) \leq \sum_{t=1}^{T} C_P(t) \leq \sum_{(i,j) \in cut(P)} \widehat{F}_i.$$

We next apply this framework to unsigned binary multiplication.

## 6 Problem: Unsigned Binary Multiplication

What computation must the case-study construction expose at its boundary? A common arithmetic and spike-train interface lets us compare internal organizations without changing the task.

We consider multiplication of two unsigned integers of equal width. For this case study, $n \geq 1$ denotes the bit width of each operand. The inputs are

$$X = \sum_{i=0}^{n-1} x_i 2^i, \qquad Y = \sum_{i=0}^{n-1} y_i 2^i, \qquad x_i, y_i \in \{0, 1\}. \tag{4}$$

Thus $0 \leq X, Y < 2^n$. The required output is their ordinary integer product

$$Z = XY = \sum_{c=0}^{2n-1} z_c 2^c, \qquad z_c \in \{0, 1\}, \tag{5}$$

where leading zeroes are retained so that every output has exactly $2n$ bit positions.

*Spike-train interface.* The system has two designated input neurons, $\sigma_X$ and $\sigma_Y$, and one designated output neuron, $\sigma_Z$. Beginning at an input time $t_0$, the operands are supplied serially, least-significant bit first. At step $t_0+i$, $\sigma_X$ receives one spike exactly when $x_i = 1$, and $\sigma_Y$ receives one spike exactly when $y_i = 1$; absence of a spike denotes zero. The input phase therefore occupies the $n$ consecutive positions $t_0, \ldots, t_0 + n - 1$.

After a construction-dependent delay, the output neuron reserves $2n$ consecutive positions. If the first output position is $t_{\text{out}}$, then $\sigma_Z$ emits one spike at $t_{\text{out}} + c$ exactly when $z_c = 1$, for $0 \leq c < 2n$. Silence inside this reserved window denotes an output zero; it does not mean that the computation has halted. Correctness therefore means that the decoded output train is the $2n$-bit, least-significant-bit-first representation of $XY$ for every $X, Y \in \{0, \ldots, 2^n - 1\}$.

This interface fixes only the problem seen at the system boundary. It does not prescribe how bits are represented or routed internally, and it is independent of the partition map $P$. With that boundary fixed, we can ask whether balanced placement alone can make the published Zhang construction inexpensive to communicate.

## 7 The Zhang Shift–Add Multiplication Unit

### 7.1 Operation of the multiplication unit

Can balanced placement make the Zhang shift–add multiplier communication-efficient? To answer, we first describe the unit presented by Zhang et al. [16] and then lower-bound its communication under any balanced placement. Figure 1 is our redraw of the diagram in their paper, with the interface labels matched to the notation used here.

Zhang et al. [16] allow operands of different bit widths and use the shorter operand as the multiplier. We use two $n$-bit operands, as fixed in Section 6. Their starting point is

$$XY = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} y_j x_i 2^{i+j} = \sum_{j=0}^{n-1} y_j X 2^j. \tag{6}$$

Each outer term is a partial product: a copy of $X$, shifted by $j$ positions when $y_j = 1$, or zero when $y_j = 0$. The construction realizes these $n$ terms in rows numbered 1 through $n$. Thus row $k$ realizes $y_{k-1} X 2^{k-1}$. The operands arrive serially, least-significant bit first: $\sigma_X$ receives $x_0, x_1, \ldots, x_{n-1}$, while $\sigma_Y$ receives $y_0, y_1, \ldots, y_{n-1}$.

In Part 1 of Fig. 1, $\sigma_X$ supplies $X$ to the $n$ partial-product rows. In row $k$, the neurons $\sigma_{ak0}, \sigma_{ak1}, \ldots, \sigma_{akk}$ apply $a \to a$ in sequence, so the copy arriving at $\sigma_{dk}$ has the displacement required by its binary position.

Part 2 supplies the corresponding bit $y_{k-1}$ of $Y$. The initially spiking $\sigma_{b0}$ sends a timing spike, or marker, through $\sigma_{b1}, \ldots, \sigma_{bn}$, while $\sigma_Y$ supplies the bits of $Y$ to the $\sigma_{ck}$ neurons. At $\sigma_{ck}$, one spike is forgotten by $a \to \lambda$, whereas two coincident spikes enable $a^2 \to a$. A firing of $\sigma_{ck}$ starts the associated $\sigma_{ek1}, \sigma_{ek2}, \sigma_{ek3}$ neurons. The delayed rule in $\sigma_{ek1}$ terminates their activity after the $n$ positions of $X$. During that interval, $\sigma_{ek2}$ supplies one spike to $\sigma_{dk}$ per position. Since $\sigma_{dk}$ forgets one spike and fires on two, it passes a bit of the displaced $X$ string exactly when the corresponding multiplier bit is one.

Part 3 sums the partial products. In one output position, let $q$ be the number of spikes in $\sigma_Z$, including spikes retained from the preceding position. If $q$ is even, the applicable rule emits no spike and leaves $q/2$ spikes. If $q$ is odd, the applicable rule emits one spike and leaves $\lfloor q/2 \rfloor$. The emitted spike is therefore the current product bit, while the spikes left in $\sigma_Z$ contribute to the next position. The product is emitted from low to high bits.

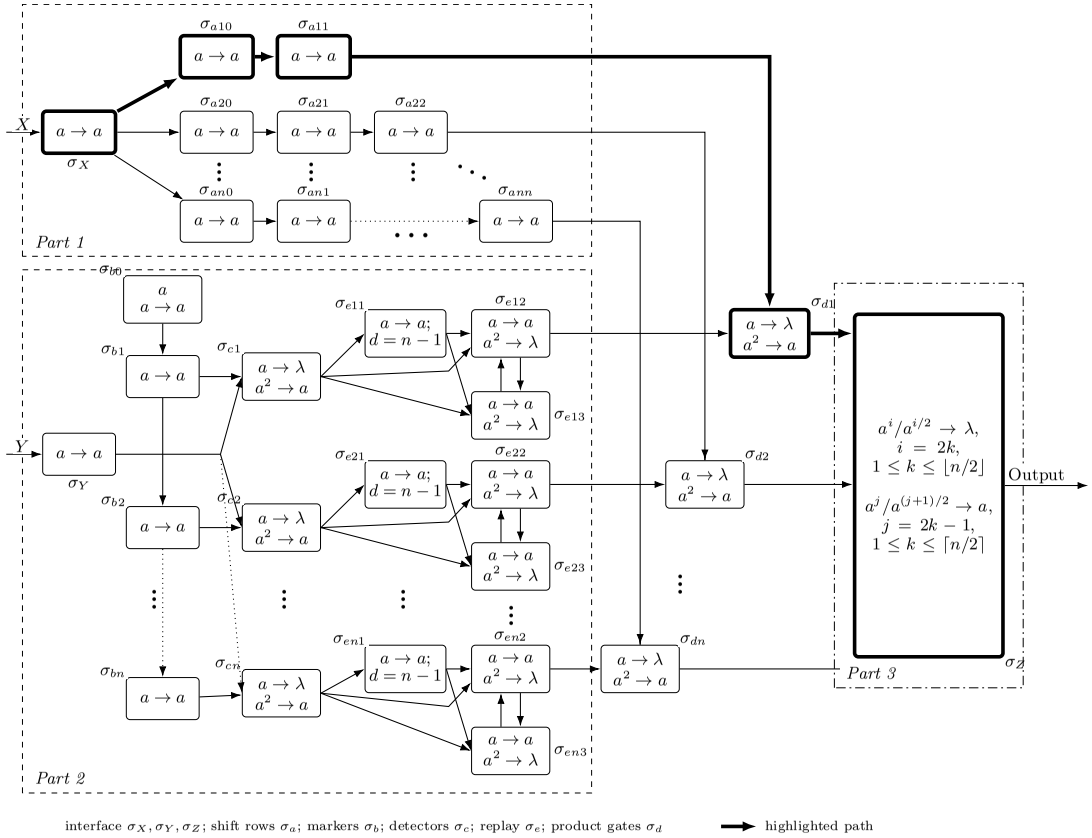



**Fig. 1.** Redraw of the multiplication-unit diagram from Zhang et al. [16]. The topology and rule labels follow the source. The interface labels match the notation used here, and the key lists the neuron families. The heavy path is $Q_1 = (\sigma_X, \sigma_{a10}, \sigma_{a11}, \sigma_{d1}, \sigma_Z)$, the $k = 1$ instance of Eq. (9) used in the analysis below.

### 7.2 Balanced-Partition Analysis

We now hold the topology and rules fixed and seek the least cumulative communication attainable by assigning their neurons to $K$ balanced blocks. Fix $K \geq 2$ independently of $n$ and let $(P_n)$ be any asymptotically balanced partition family. We analyze the all-one input

$$X = Y = (\underbrace{11\cdots1}_{n \text{ bits}})_2 = 2^n - 1, \tag{7}$$

because it activates every partial-product row and produces the most communication along the paths used in the proof. It therefore represents the worst case studied here. We use the cumulative communication $H(P_n)$ from Definition 4.

To apply Definition 5, we use the neuron count reported by Zhang et al. [16]:

$$N_Z(n) = \frac{(n+1)(n+2)}{2} + (5n+2) + n + 1 = \frac{n^2 + 15n + 8}{2}. \tag{8}$$

The three terms count Parts 1 and 2 and the neurons $\sigma_{d1}, \ldots, \sigma_{dn}, \sigma_Z$, respectively.

For the analysis only, let $Q_k$, $1 \leq k \leq n$, denote the directed path

$$Q_k = (\sigma_X, \sigma_{ak0}, \sigma_{ak1}, \ldots, \sigma_{akk}, \sigma_{dk}, \sigma_Z). \tag{9}$$

The construction contains one $\sigma_X$ and one $\sigma_Z$. Every $Q_k$ uses these two neurons as its endpoints, and a partition assigns each endpoint to exactly one block. The paths share no other neurons or synapses; Fig. 1 highlights $Q_1$. Traffic on this subgraph already lower-bounds total communication, so paths carrying $Y$ are unnecessary for the proof.

**Lemma 1 (All-one input activity).** *On the input in Eq.* (7)*, every edge of every $Q_k$ carries exactly $n$ spike transmissions.*

*Proof.* The $n$ spikes encoding $X$ traverse $\sigma_X$ and every $\sigma_a$ neuron on $Q_k$. Because every bit of $Y$ is one, $\sigma_{dk}$ is enabled for all $n$ positions and passes all $n$ spikes to $\sigma_Z$. The longer chains change arrival times, not the number of transmissions per edge. □

**Proposition 3 (Balanced-partition communication lower bound).** *On the all-one input, every asymptotically balanced family $(P_n)$ satisfies $H(P_n) = \Omega(n^2)$. More precisely,*

$$H(P_n) \geq \frac{1}{K}\left(1 - \frac{1}{\sqrt{K}}\right) n^2 - o(n^2) \tag{10}$$

*Proof.* Call $Q_k$ *block-local* if all its neurons lie in one block. Let $S_n$ be the set of indices $k$ for which $Q_k$ is block-local, and define $u_n = |S_n|$, the number of block-local paths. If $\sigma_X$ and $\sigma_Z$ lie in different blocks, then $S_n$ is empty and $u_n = 0$. If both endpoints lie in the same block $B$, every path indexed by $S_n$ must also lie

in $B$. Apart from the two shared endpoints, $Q_k$ contains $k+2$ neurons: $k+1$ of type $\sigma_a$ and $\sigma_{dk}$. These neurons are disjoint across paths. Because $k+2$ increases with $k$, the quantity $\sum_{k\in S_n}(k+2)$ is minimized among all index sets of size $u_n$ when $S_n = \{1, \ldots, u_n\}$. Therefore,

$$|B| \geq 2 + \sum_{k\in S_n}(k+2) \geq 2 + \sum_{k=1}^{u_n}(k+2) = 2 + \frac{u_n^2 + 5u_n}{2}.$$

By the balance requirement in Definition 5, whose bound is Eq. (3), block $B$ can contain at most $N_Z(n)/K + o(N_Z(n))$ neurons. Equation (8) gives $N_Z(n) = n^2/2 + O(n)$, so

$$|B| \leq \frac{n^2}{2K} + o(n^2).$$

On the other hand, the preceding path count gives $|B| \geq u_n^2/2$. Combining these two bounds yields

$$\frac{u_n^2}{2} \leq \frac{n^2}{2K} + o(n^2), \qquad \text{and hence} \qquad u_n \leq \frac{n}{\sqrt{K}} + o(n).$$

The same bound holds when $\sigma_X$ and $\sigma_Z$ are in different blocks, because then $u_n = 0$. Of the $n$ paths $Q_1, \ldots, Q_n$, exactly $u_n$ are block-local. Each of the remaining $n - u_n$ paths spans more than one block and therefore contains a cut synapse. Using the bound on $u_n$, their number satisfies

$$n - u_n \geq \left(1 - \frac{1}{\sqrt{K}}\right) n - o(n).$$

Choose one cut synapse from each cut path. These synapses are distinct because the paths share no synapses. By Lemma 1, each chosen synapse carries $n$ transmissions. Therefore,

$$\sum_t C_{P_n}(t) \geq n(n - u_n) \geq \left(1 - \frac{1}{\sqrt{K}}\right) n^2 - o(n^2).$$

Thus, the paths $Q_1, \ldots, Q_n$ alone force $\Omega(n^2)$ cumulative cross-block spike transmissions. Equation (1) converts this aggregate traffic into a lower bound on the per-step BSP communication bottleneck:

$$H(P_n) = \sum_t h(t) \geq \frac{1}{K}\sum_t C_{P_n}(t) \geq \frac{1}{K}\left(1 - \frac{1}{\sqrt{K}}\right) n^2 - o(n^2),$$

which proves Eq. (10). □

Thus, even the best balanced placement requires $\Omega(n^2)$ communication on the all-one input. This does not establish an average-input lower bound. The cause is the set of long paths that remain active throughout the input and share the same endpoints. This shows what a redesign should avoid and motivates the next section.

## 8 The Carry-Save SN P Multiplier

If balanced placement cannot remove the Zhang multiplier's quadratic communication on the all-one input, what should change in the multiplier itself? The analysis in Section 7 motivates replacing the lengthening shift paths with temporal reuse and the large convergence points with small reduction components. The structure-aware criterion in Section 5 adds a third requirement: expose a spatial order that a partition can follow. Figure 2 illustrates one response centered on a hierarchical carry-save reducer. After describing it, we apply the communication part of the BSP framework to its intended schedule.

The sketch uses $n$ repeated lane modules, one for each bit $x_i$ of $X$. Lane $i$ keeps $x_i$ available while the bits of $Y$ pass through it, thereby forming the partial products $x_i y_j$ over time. Following the standard carry-save reduction principle [13], a reduction network repeatedly compresses three aligned lane streams into two without propagating their carries through the entire result; a final accumulator then emits the product serially. This is the construction's carry-save mechanism. Three-to-two reduction has previously been used to build SN P iterated-addition subsystems for Subset Sum [7]; here, it is organized as a streamed multiplier whose lane order is intended to support partitioning. The lane index records where a partial product is formed, while the time slot records its product position.

*Part 1: select and replay $X$.* The serial $X$ stream enters $\sigma_X$ and is distributed to one detector $\sigma_{ci}$ per lane by an equal-depth relay tree. Its depth $d = \lceil \log_2 n \rceil$ gives every root-to-detector route the same forwarding delay. A marker $\sigma_{bi}$ is timed to meet bit $x_i$ at its detector. If $x_i = 1$, the two coincident spikes start the feedback pair $\sigma_{ei2}, \sigma_{ei3}$; if $x_i = 0$, the detector forgets the lone marker spike. A delayed spike from $\sigma_{ei1}$ stops an active feedback pair after the $n$ positions of $Y$. The selected bit is therefore replayed beside its lane instead of sending $X$ through a separate, lengthening shift path for every partial-product row.

*Part 2: move $Y$ through the lanes.* After an alignment prefix of $d + 2$ relays, the bits of $Y$ advance through the ordered lane relays $\sigma_{y0}, \ldots, \sigma_{y(n-1)}$. In lane $i$, the partial-product gate $\sigma_{di}$ receives the replayed $x_i$ and the passing $y_j$; it emits exactly when both bits are one. The resulting lane stream is

$$P_i = (x_i y_j)_{j=0}^{n-1}. \tag{11}$$

Let $\tau$ be the common offset introduced by input alignment. The intended schedule places $x_i y_j$ in slot $t_{i,j} = \tau + i + j$. Hence all terms with the same product index $c = i + j$ meet the reduction network in slot $\tau + c$. The heavy route in Fig. 2 traces this flow for $P_1$.

*Part 3: carry-save reduction.* The reduction network repeatedly replaces three aligned streams by two. For three bits $u, v, w$, the arithmetic identity

$$u + v + w = (u \oplus v \oplus w) + 2\ \mathrm{maj}(u, v, w)$$

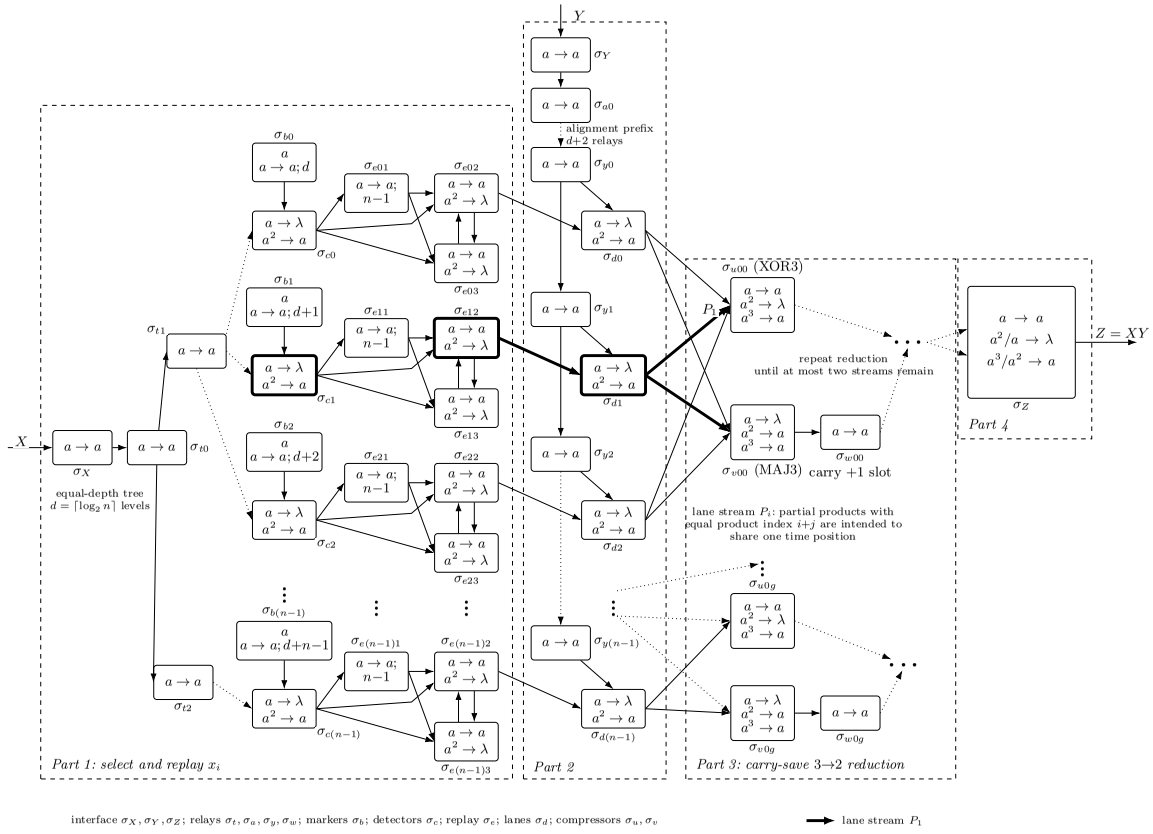



**Fig. 2.** Design sketch of an $n$-lane carry-save SN P multiplier. The dashed regions show $X$-bit selection and replay, the $Y$ pipeline, carry-save reduction, and serial accumulation. Repeated structure is omitted; heavy lines trace $P_1$ from Eq. (11).

explains the two outputs. The three-input exclusive-or neuron (XOR3) produces the sum bit in the current product position. Here $\mathrm{maj}(u, v, w) = 1$ exactly when at least two inputs are one, so the three-input majority neuron (MAJ3) produces the carry. A relay delays that carry by one slot, moving it to the next product position. At each stage, consecutive streams in lane order are grouped in threes. The same operation repeats until at most two streams remain; delayed relays keep any ungrouped stream aligned with the next stage. If $h_0 = n$ and

$$h_{r+1} = 2\left\lfloor \frac{h_r}{3} \right\rfloor + (h_r \bmod 3),$$

then the number $R$ of stages before $h_R \leq 2$ is $O(\log n)$. Thus, no reduction stage requires all $n$ partial-product streams to converge on one neuron: each compressor receives only three streams.

*Part 4: emit the product.* The accumulator $\sigma_Z$ receives the remaining streams and retains their binary carry from one position to the next. Its parity and carry rules emit one product bit per slot, from least to most significant. The proposed architecture is therefore intended to preserve the serial multiplication interface fixed in Section 6, while changing how the partial products reach the output.

## 9 Preliminary Lane-Band Analysis

We analyze the sketched topology using an ordered binary broadcast tree whose nodes cover consecutive lane intervals and the intended schedule described above.

Give every marker, detector, replay neuron, $Y$ relay, and partial-product gate of lane $i$ coordinate $i$. A broadcast-tree neuron receives the lower median coordinate of its descendant lanes, and $\sigma_X$ receives the root's coordinate. At a reduction stage, a compressor and its carry relay receive the median coordinate of their three source streams, while a leftover relay keeps its source coordinate. The alignment prefix and $\sigma_Y$ receive coordinate zero, and the accumulator receives coordinate $n-1$. Let $P_n^{\mathrm{lane}}$ divide consecutive coordinates among the $K$ blocks.

*Preliminary lane-band estimate.* For every fixed $K \geq 2$ and all sufficiently large $n$, the lane coordinates can be divided into an asymptotically balanced family $(P_n^{\mathrm{lane}})$. For the sketched topology and schedule, the boundary and activity counts below give, for every input, the preliminary estimate

$$H(P_n^{\mathrm{lane}}) \leq (K-1)\big(2dn + n + 2n(2R+2)\big) = O(Kn \log n). \tag{12}$$

Thus, $H(P_n^{\mathrm{lane}}) = O(n \log n)$ when $K$ is fixed.

*Justification.* Let $N_S(n)$ count the neurons in the proposed topology. The $n$ lane modules contribute $\Theta(n)$ neurons. The broadcast tree has $O(n)$ neurons, and the stream counts $h_r$ decrease geometrically, so all reduction stages together also use $O(n)$ neurons. Hence $N_S(n) = \Theta(n)$.

Each coordinate contains $O(\log n)$ neurons: a constant-size lane module, at most one broadcast neuron per level, and, per reduction stage, either the three neurons of at most one compressor group or at most two leftover relays. At stage zero, each coordinate occurs once. Grouping the ordered coordinates in threes preserves their order. A median is copied twice, but another copy outside its group would require at least three input copies, so no coordinate occurs more than twice at later stages. The alignment prefix adds $O(d)$ neurons at coordinate zero. Since $d, R = O(\log n)$, dividing the ordered coordinates at cumulative sizes nearest $bN_S(n)/K$, $1 \le b < K$, gives each block $N_S(n)/K + O(\log n)$ neurons and is asymptotically balanced.

Fix one boundary between consecutive coordinate bands. Lane-local synapses do not cross it, and the ordered $Y$ pipeline contributes one crossing synapse. At one broadcast level, at most one descendant interval spans the boundary, so at most two broadcast synapses cross. At one reduction stage, at most one consecutive triple spans the boundary. Placing its compressor at the median leaves at most one source on the other side, giving at most two crossing synapses. The final streams add at most two more. The boundary therefore crosses at most one $Y$ synapse, $2d$ broadcast synapses, and $2R + 2$ reduction or final-stream synapses.

Every broadcast or $Y$ synapse carries at most $n$ spike transmissions. A reduction or final-stream synapse carries at most one spike in each of fewer than $2n$ product positions. Summing these bounds over the $K - 1$ boundaries gives the first inequality in Eq. (12); an edge spanning several boundaries is only overcounted. Finally, Eq. (1) gives $H(P_n^{\mathrm{lane}}) \le \sum_t C_{P_n^{\mathrm{lane}}}(t)$, which yields the stated estimate.

On the all-one input, Proposition 3 gives $H(P_n) = \Omega(n^2)$ for every balanced placement of the Zhang construction, whereas Eq. (12) gives a balanced $O(n \log n)$ estimate for the sketched topology and schedule. This preliminary comparison covers cumulative communication. The sketch also uses fewer neurons. Equation (8) gives $N_Z(n) = \Theta(n^2)$ for the Zhang unit, while the sketch uses $N_S(n) = \Theta(n)$. Completing the construction will establish its correctness and supply the local-work and timing bounds needed for the total cost in Definition 4. Equation (12) supplies the communication estimate for the structure-aware family in Definition 7. Evaluating the relative saving $R_n(P', P)$ also requires a named structure-blind baseline family for the carry-save system.

## 10 Conclusion and Research Outlook

For constructions intended for distributed execution, our results support treating partitionability as a property of an SN P construction, not solely as an after-the-fact placement decision. As formalized by Proposition 1, the distributed execution induced by a static neuron-to-block map has the same possible computations and outputs as the original SN P system, while the cost of that execution depends jointly on the construction's topology, spike activity, and that

map. When distributed execution is a design goal, this interaction motivates using communication analysis to inform both construction and placement.

This paper made that interaction explicit through a partitioned model, a placement-invariance result, a BSP cost, and the partition-aware criterion of Definition 7. In the Zhang shift–add multiplier, the all-one input keeps long paths with shared endpoints active throughout the input window. Consequently, on that input, Proposition 3 gives the quadratic lower bound $H(P_n) = \Omega(n^2)$ for every asymptotically balanced partition into a fixed number of blocks. For the proposed carry-save topology and intended schedule, Section 9 gives the preliminary estimate $H(P_n^{\text{lane}}) = O(n \log n)$ for a balanced lane-band placement and fixed $K$. This supplies a preliminary communication-side comparison between the diagnosed construction and the redesign. The sketched redesign is also smaller, using $\Theta(n)$ neurons against $\Theta(n^2)$.

*Completing the carry-save construction.* The immediate next step is to develop the sketch in Section 8 into a complete SN P construction. This requires a neuron- and rule-level specification, a correctness proof for the serial multiplication interface, and resource and timing analyses. The lane-band estimate concerns cumulative communication, but local work and total BSP cost must still be compared with the Zhang construction under the same interface and computation family. Definition 7 also requires a named structure-blind baseline family for the carry-save system.

*Simulation and large-instance scaling.* The analysis should be paired with empirical validation. Small instances can first check the output stream, the intended schedule, and per-step cross-block traffic against the analytical model. We then plan a distributed capacity experiment following the capacity-scaling approach used in our earlier MPI+CUDA study [9]: increase the operand width until one accelerator exhausts its memory, then test whether multiple nodes can simulate larger instances without communication dominating execution. The Zhang and carry-save constructions should be compared at the same operand width and on the same inputs. The evaluation will report the maximum operand width, neuron, synapse, and rule counts, peak host and device memory, cross-block traffic, execution time, and strong and weak scaling. Allowing the number of blocks to grow with the operand width will also probe a regime beyond the fixed-$K$ lower bound established here.

*Testing the cost model against SN dP systems.* An SN dP system already carries a decomposition into cooperating SN P modules [5]. Flattening one into a single SN P system and partitioning it under our cost model would ask whether the cheapest placement recovers the original decomposition. The two models distribute input differently and balance different quantities, so the comparison first requires defining a faithful flattening.

*Beyond multiplication.* The longer-term objective is to apply the same workflow to other SN P constructions: identify the active routes responsible for communication, redesign their topology or schedule, and expose coordinates that a

structure-aware partition can follow. This also suggests automated tools that determine which neurons may fire, bound their emitted spikes over a stated input family and time window, and combine those bounds with candidate cuts. Such tools could estimate communication before execution and help compare candidate placements. More broadly, distributed cost should become a construction-time concern, so that an SN P system's topology, schedule, and placement are designed together rather than optimized separately after the fact.

**Acknowledgement.** F. G. C. Cabarle is supported by the 2024–2026 Scientific Productivity System of the University of the Philippines (UP), the San Francisco and Mely and Rick Rey Foundation Professorial Chair of the UP Alumni Association, and RLC from the UP Diliman OVCRD. This work is part of Project PID2025-174204NB-I00, funded by the Ministry of Science, Innovation and Universities (MICIU), the State Research Agency (AEI), DOI 10.13039/501100011033, and the European Regional Development Fund (ERDF), European Union. This work is part of a project that has received funding from the European Union's Horizon Europe Research and Innovation Programme under the Marie Skłodowska-Curie Actions (MSCA), Grant Agreement No. 101236749. Thank you very much to Mario J. Pérez-Jiménez who is an inspiration of our investigations and the topics in our investigations.

## References

1. Martínez-del Amor, M.Á., Orellana-Martín, D., Pérez-Hurtado, I., Cabarle, F.G.C., Adorna, H.N.: Simulation of spiking neural P systems with sparse matrix-vector operations. Processes **9**(4), 690 (Apr 2021). https://doi.org/10.3390/pr9040690
2. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent Advances in Graph Partitioning, pp. 117–158. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-49487-6_4
3. Cremonesi, F., Schürmann, F.: Understanding computational costs of cellular-level brain tissue simulations through analytical performance models. Neuroinformatics **18**(3), 407–428 (Feb 2020). https://doi.org/10.1007/s12021-019-09451-w
4. Fernandez-Musoles, C., Coca, D., Richmond, P.: Communication sparsity in distributed spiking neural network simulations to improve scalability. Frontiers in Neuroinformatics **13** (Apr 2019). https://doi.org/10.3389/fninf.2019.00019
5. Ionescu, M., Păun, G., Pérez-Jiménez, M.J., Yokomori, T.: Spiking neural dP systems. Fundamenta Informaticae **111**(4), 423–436 (Aug 2011). https://doi.org/10.3233/fi-2011-571
6. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae **71**(2-3), 279–308 (Jan 2006). https://doi.org/10.3233/fun-2006-712-308
7. Leporati, A., Gutiérrez-Naranjo, M.A.: Solving SUBSET SUM by spiking neural P systems with pre-computed resources. Fundamenta Informaticae **87**(1), 61–77 (2008), https://journals.sagepub.com/doi/abs/10.3233/FUN-2008-87105
8. Leporati, A., Mauri, G., Zandron, C.: Spiking neural P systems: main ideas and results. Natural Computing **21**(4), 629–649 (Aug 2022). https://doi.org/10.1007/s11047-022-09917-y

9. Lipat, J., Luzada, J.S., Cabarle, F.G.C., Martínez-del Amor, M.Á., Orellana-Martín, D.: Distributed SNP: A high-performance hybrid MPI+CUDA framework for spiking neural P systems. In: Azcarraga, J.J., Sityar, L. (eds.) Proceedings of the 26th Philippine Computing Science Congress (PCSC 2026). pp. 77–84. Computing Society of the Philippines, Davao City, Philippines (2026), https://pcsc2026.uic.edu.ph/papers
10. McColl, W.F.: Scalable computing, pp. 46–61. Springer Berlin Heidelberg (1995). https://doi.org/10.1007/bfb0015236
11. Muniyandi, R.C., Maroosi, A.: A representation of membrane computing with a clustering algorithm on the graphical processing unit. Processes **8**(9), 1199 (Sep 2020). https://doi.org/10.3390/pr8091199
12. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8), 103–111 (Aug 1990). https://doi.org/10.1145/79173.79181
13. Wallace, C.S.: A suggestion for a fast multiplier. IEEE Transactions on Electronic Computers **EC-13**(1), 14–17 (Feb 1964). https://doi.org/10.1109/pgec.1964.263830
14. Zeng, X., Adorna, H., Martínez-del Amor, M.Á., Pan, L., Pérez-Jiménez, M.J.: Matrix Representation of Spiking Neural P Systems, pp. 377–391. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-18123-8_29
15. Zeng, Y., Feng, G., Chen, Z., Lu, Y., Xiao, N.: ATM: Area-based partition and topology-aware mapping for large-scale SNN simulation. In: 2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA). pp. 1841–1848. IEEE (Oct 2024). https://doi.org/10.1109/ispa63168.2024.00251
16. Zhang, G., Rong, H., Paul, P., He, Y., Neri, F., Pérez-Jiménez, M.J.: A complete arithmetic calculator constructed from spiking neural P systems and its application to information fusion. International Journal of Neural Systems **31**(01), 2050055 (Sep 2020). https://doi.org/10.1142/s0129065720500550

# Small universal multiset reaction systems

Annemarie-Beatrix Messner[1] and Andrei Păun[1,2,3]

[1] Faculty of Mathematics and Computer Science, University of Bucharest, Bucharest, Romania
annemarie-beatrix.messner@s.unibuc.ro, apaun@fmi.unibuc.ro
[2] National Institute of Research and Development for Biological Sciences, 296 Independenței Bd. District 6, 060031, Bucharest. Romania
andrei.paun@incdsb.ro
[3] Research Institute for Artificial Intelligence (ICIA), Romanian Academy, Bucharest, Romania.
apaun@racai.ro

**Abstract.** This paper presents a construction of a small universal machine within the framework of Multi-set Reaction Systems. We encode the machine state using elements that represent registers and instruction labels, and we enforce sequential execution by ensuring that reaction chains for a given instruction can only activate after the corresponding label element appears. We implement a universal register machine with 8 registers and 23 instructions, showing that the priority-based model requires 87 distinct elements. The results demonstrate that multiset reaction systems are capable of simulating universal computation.

**Keywords:** Reaction · Reaction system · Interactive process · multiset reaction system · P systems

## 1 Introduction

Reaction Systems (RS), introduced by Andrzej Ehrenfeucht and Grzegorz Rozenberg [1, 2], provide a qualitative framework for modeling biochemical and regulatory processes based on the fundamental mechanisms of facilitation and inhibition. In this formalism, the state of the system is represented as a set of entities, and reactions are enabled solely by the presence or absence of these entities. While this abstraction successfully captures the logical structure of regulatory interactions, it deliberately neglects quantitative aspects such as molecular concentrations, multiplicities, and reaction rates. In living cells, molecular species are typically present in multiple copies, and their abundance plays a decisive role in determining whether reactions occur and at what rate. The assumption that an entity is either present or absent ignores the influence of concentration on system dynamics and prevents the description of many important biological phenomena, including threshold effects, competition for limited resources, and stochastic fluctuations arising from finite molecular populations.

Introducing multisets into the framework naturally overcomes this limitation. A multiset associates each entity with a multiplicity, allowing the state

of the system to represent molecular counts rather than mere existence. Reactions can then consume and produce specified numbers of molecules, providing a more faithful representation of biochemical transformations while preserving the structural simplicity of the reaction-system paradigm. This extension bridges the gap between purely qualitative models and quantitative models based on molecular populations.

A further step toward realism consists in enriching reactions with kinetic information. Classical Reaction Systems treat all enabled reactions as occurring simultaneously and without regard to their relative speeds. In contrast, chemical and biochemical reactions proceed at rates determined by intrinsic kinetic constants and by the concentrations of the reacting species. Associating a kinetic parameter with each reaction allows the evolution of the system to reflect not only the logical conditions for reaction occurrence but also the temporal scale on which reactions take place.

This quantitative extension naturally connects the framework to the law of mass action, one of the cornerstones of chemical kinetics. According to the law of mass action, the rate of an elementary reaction is proportional to the product of the concentrations (or molecular counts in a discrete setting) of its reactants, each raised to the power of its stoichiometric coefficient. By combining multisets with reaction kinetics, the enabling of reactions remains governed by the presence of reactants and the absence of inhibitors, as in the original Reaction Systems framework, while the frequency or probability of reaction execution becomes proportional to the corresponding mass-action rate. Consequently, the resulting formalism preserves the regulatory semantics that characterize Reaction Systems while incorporating the quantitative behavior that governs real biochemical processes.

This enriched framework therefore provides a natural bridge between qualitative regulatory models and quantitative kinetic models. It retains the conceptual elegance and compositional nature of Reaction Systems while extending their expressive power to describe concentration-dependent dynamics, stoichiometric constraints, and time-dependent behavior. Such an approach enables the study of biological systems at a level of abstraction that remains mathematically tractable yet is substantially closer to the mechanisms underlying real chemical and biochemical reactions.

We provide in the following a step in this direction by considering the multiset RS and changing the behaviour of the system in the sense that the reaction with most reactants will fire next in a sequential manner.

Multiset reaction systems, formally defined in [3], introduce the use of multisets instead of sets to represent the reactants, inhibitors and products of a reaction. This allows the model to be quantitative, as it can now take into account the quantity of a resource. The permanency principle is modified to allow substances that are not consumed to persist to the next state. This results in a model that is inherently nondeterministic as the reactions will have to compete for the available resources in a state. To model a specific multiset reaction sys-

tem, we need to define the background set of elements and the set of possible reactions.

Small universal register machines were described in [4] and have been used as a standard model of computation to show the Turing completeness of various computational models in the membrane systems area, such as spiking neural P systems [5] and [6]. In the following we will use simiar ideas from the aforementioned papers to the area of reaction systems. In this paper we will show the implementation of a small universal register machine with 23 instructions using multiset reactions systems using max-spiking manner. This involves the modelling of the ADD, SUB and HALT instructions using reactions, as well as encoding the instruction labels and registers as elements in the background set. We prove that this approach is universal and give an upper bound for the number of components of the system.

## 2 Preliminaries

We first recall the basic notions concerning reaction systems, introduced in [1, 2]. Given a finite and nonempty set $S$, we say that a *reaction* in $S$ is a triplet $a = (R, I, P)$, where $R, I, P$ are subsets of $S$ and $R \cap I = \emptyset$. The elements of $S$ are informally called objects, the elements of $R$ are called *reactants*, the elements of $I$ are called *inhibitors*, while the elements of $P$ are called *products*.

A *reaction system* in $S$ is an ordered pair $\mathcal{A} = (S, A)$, where $A$ is a nonempty set of reactions in $S$. We now define the conditions for a reaction in $S$ to be enabled. Given the reaction system $\mathcal{A} = (S, A)$ and a subset $X$ of $S$, the reaction $a = (R_a, I_a, P_a)$ is enabled by $X$, denoted by the boolean function $en_a(X)$, if $R_a \subseteq X$ and $I_a \cap X = \emptyset$ holds. The result of $a$ on $X$, denoted by $res_a(X)$, is a set defined by $res_a(X) = P_a$ if $en_a(X)$, and $res_a(X) = \emptyset$, otherwise. Clearly, $X \subseteq S$ may enable more than one reaction, and this is captured by the following definition. Let $\mathcal{A} = (S, A)$ be a reaction system with background set $S$ and let $X \subseteq S$. The subset of reactions of $A$ enabled by $X$, denoted by $en(A, X)$, is defined as $en(A, X) = \{a \in A | en_a(X)\}$, and the result of $A$ on $X$, denoted by $res(A, X)$, is $res(A, X) = \bigcup_{a \in en(A,X)} res_a(X)$. As one can see, the effect of a set of reactions enabled by a subset of $S$ is cumulative. In what follows, we say that $X \subseteq S$ is a state of the system.

The dynamics of reaction systems is determined by the interactions of successive reactions enabled by successive states. Thus, the system starts in an initial state enabling a set of reactions that produce a result which will be enhanced by some context and form the new state. More precisely, the dynamics of a reaction system is described by an *interactive process* as follows. Let $\mathcal{A} = (S, A)$ be a reaction system, an interactive process in $\mathcal{A}$ is a pair of two finite sequences, $\pi = (\alpha, \beta)$, where $\alpha = C_0, C_1, \ldots, C_n$ is the *context sequence*, $\beta = D_1, D_2, \ldots, D_n$ is the *result sequence*, with $n \geq 1$, and $C_i, D_j \subseteq S$, $0 \leq i \leq n$, $1 \leq j \leq n$. The components of the context sequence are given and they represent the influence of the environment on the computation, while

the components of the result sequence are computed by $D_1 = res(A, C_0)$, and $D_i = res(A, D_{i-1} \cup C_{i-1})$ for each $2 \leq i \leq n$.

## 3 Multiset Reaction Systems

In this model, a reaction is formally defined as a triple $(R, I, P)$ where $R$ is the *set of reactants* $I$ is the *set of inhibitors*, and $P$ is the *set of products*. Two fundamental principles govern the execution of reactions: *activation* and *inhibition*. A reaction is enabled when all of its reactants are present in the current state, but none if its inhibitors are. As a result, the enabled reaction produces a set of products in the next state that could either activate or inhibit other reactions.

The original reaction systems model is considered a qualitative model of biochemical interactions, as it does not take into account the quantity of a resource. If a reactant is available in a state, it is available in an infinite quantity. On the other hand, if an element is consumed in a reaction, then it is completely removed from the state. In this model, elements cannot persist between states if they are not produced by a reaction.

Furthermore, the original reaction systems model is a computation state-transition system in which successor states are generated by previously produced reactants and input from the environment. This makes the model *context-dependent*. A model that does not consider the contribution of the environment is known as *context-independent*.

Multiset Reaction Systems are a context-independent extension of the original reaction systems model that incorporates the quantity of a resource. The model is defined in [3] and is based on replacing the sets of reactants, inhibitors, and products with multisets. This allows the modification of the permanency principle. In this model, resources can persist between states if they are not entirely consumed by reactions. As each resource is available in a specific quantity, a constraint is effectively placed on the number of reactions that can be executed in a state. This results in an inherently quantitative and nondeterministic model. Now we will give the definitions of a few concepts related to multiset reaction systems.

Multiset reaction systems are formally defined in [3] as a pair $\mathcal{A} = (S, A)$ where $S$ is a finite set of elements called the *background set* and $A$ is a finite set of reactions. $S$ is a finite set of at least two elements and its components are informally known as *objects*. The elements of $A$ are called *reactions* and are defined as triplets $(R, I, P)$ where $R$, $I$, and $P$ are multisets over $S$ representing the *reactants*, *inhibitors*, and *products* of the reaction, respectively. $R$ and $I$ are disjoint multisets, meaning that they cannot share any common elements. We denote as $MRS(S)$ the set of all multiset reaction systems over the background $S$.

Multisets extend the definition of sets by allowing each element to be accompanied by a multiplicity, which indicates how many times that element appears in the multiset. In order to compute the number of copies for each element, we

employ a *membership function* $\sigma_X : U \to \mathbb{N}$, where $\sigma_X(x)$ denotes how many times the element $x$ from a finite set $U$ appears in the multiset $X$. An *empty multiset* is a multiset in which all elements have a multiplicity of zero and is known as $\epsilon$. The *weight* of a multiset $X$ over $U$ is defined as:

$$\|X\| = \sum_{a \in U} \sigma_X(a).$$

A reaction $a = (R, I, P)$ is *enabled (activated)* by a state $T$ if $R$ is a submultiset of $T$ and $I$ is disjoint from $T$. When a reaction is enabled, it produces the multiset of products $P$ in the next state. We note whether a reaction $a$ is enabled by a state $T$ as $\text{en}_a(T)$. A key difference between the original reactions systems model and the multiset reaction systems model is that elements that are not removed as a result of a reaction can persist in the next state if not consumed. This helps us define $\text{res}_a(T)$, the result of executing a reaction $a$ in a state $T$ as:

$$\text{res}_a(T) = \begin{cases} P + (T \setminus R) & \text{, if } \text{en}_a(T) \\ \emptyset & \text{, otherwise} \end{cases}$$

For a multiset reaction $\Delta$ in a multiset reactions system $\mathcal{A} = (S, A)$, we define $R_\Delta$, $I_\Delta$ as follows:

$$R_\Delta = \sum_{a \in A} \sigma_\Delta(a) \cdot R_a$$

$$I_\Delta = \bigcup_{\substack{a \in A \\ \sigma_\Delta(a) \geq 1}} I_a$$

The multiset reaction systems model defines a state transition system over $S$ with a computation path $T_0, T_1, T_2, ...$ where $T_{i+1} = \bigcup_{a \in A} \text{res}_a(T_i)$ for all $i \geq 0$. We consider that a state $T$ enables simultaneously the reactions of $\Delta$, noted as $\text{en}_\Delta(T)$, if $R_\Delta$ is a submultiset of $T$ and $I_\Delta$ is disjoint from $T$. The result of executing a multiset of reactions is defined as:

$$\text{res}_\Delta(T) = \sum_{a \in A} \sigma_\Delta(a) \cdot P_a + (T \setminus R_\Delta)$$

If there is a multiset of reactions $\Delta$ of weight $k$, such that $\text{en}_\Delta(T)$ is True, then we call that a *k-event*. A *1-event* is also known as a *sequential event*.

An *f-evolution* generated by $T$ is a sequence of multisets over S: $D_0, D_1, D_2, ...$ such that $D_0 = T$ and for all $i \geq 0$, the multiset $D_{i+1}$ is generated from $D_i$ by and *f-event* such that: $D_{i+1} = \text{res}_{\Delta_i}(D_i)$.

The computation of the machine $M_a \in MRS(S)$ where $a \in S$ is given by the multiplicity of the element $a$ in the halting position of the machine $M_a$. If the machine does not halt we do not consider that computation to genereate any output. Thus $L(M_a) = \{\sigma_X(a) \mid X \text{ is the halting state of the machine}\}$.

For computing multiset reaction systems, a special *universal inhibitor*, noted as †, is introduced and considered to be a part of the inhibitor set of every reactions. Consequently, if † is present in a state, then no reaction can be enabled and the system halts.

### 3.1 The register machines

The register machine is a well-known computational model, composed of a set of registers and a finite set of instructions. Each of these register can store a non-negative integer value. The machine operates by manipulating these values through a sequence of arithmetic operations and control flow branches.

Formally, the register machine is defined as $M = (m, H, L_0, L_h, I)$ where:

- $m$ is the number of registers, which are denoted as $R_0, R_1, ..., R_{m-1}$,
- $H$ is the finite set of instruction labels,
- $L_0 \in H$ is the initial instruction label,
- $L_h \in H$ is the halting instruction label,
- $I$ is a finite set of instructions, that are bijectively mapped to $H \setminus \{L_h\}$.

A register machine $M$ starts with all registers initialized to zero and proceeds to generate a set of number. First the instruction labeled $L_0$ is applied, and then the machine proceeds to execute instructions as defined by the labels. If the instruction $L_h$ is reached, the machine halts and the final state of the register $R_0$ is considered the output of the computation. If the machine does not halt, then no output is generated. All the numbers generated by a register machine $M$ compose the set denoted as $\mathcal{N}(M)$. All the numbers generated by a register machine are known to be Turing computable [7].

All functions that can be computed by a Turing machine can also be computed by register machines. For this we need to introduce the arguments in specific registers: $R_1, R_2, ..., R_k$ and then start the execution by running the instruction with the label $L_0$. When the instruction labeled $L_h$ is reached and the machine halts, then the value of the function can be found in the register $R_0$.

In [4], the register machines are used for computing functions. In this context, a register machine is defined as universal if there is a recursive function $g$ such that for all natural numbers $x$, $y$ we have $\phi_x(y) = M_u(g(x), y)$, where $(\phi_0, \phi_1, ...)$ are a fixed admissible enumaration of the unary partial recursive functions [4].

Several strongly universal register machines are constructed in [4], having the the numbers $g(x)$ and $y$ introduced as input in registers $R_1$ and $R_2$. The result is obtained in register $R_0$.

## 4 Small universal systems

Multiset Reaction Systems are universal in the max-sequentiality mode. To prove this we will show that any Turing Machine working of numbers can be simulated by a MRS.

**Theorem 1.** *Multiset Reaction Systems are universal in the max-sequentiality mode.*

*Proof.* The proof proceeds by simulating the set $T$ generated by the TM through a register machine, thus we start by encoding the registers and instructions labels as elements in the background set and then constructing the instruction set for

a small non-deterministic universal machine as chains of reactions, effectively showing how each instruction can be represented within the multiset reaction system formalism. Given the register machine we consider the element $R_0$ as the encoding of the register 0, thus we will prove that there is $M_{R_0} \in MRS(S)$ where $R_0 \in S$ that for the element $n \in T$ will have the value $2n$ in the halting computation of the machine $M_{R_0}$.

The instructions we consider are:

$L_i$ : ADD$(R_i, L_j, L_k)$ - Add 1 to the value stored in register $R_i$, then jump non-deterministically to either instruction with label $L_j$ or to instruction with label $L_k$.

$L_i$ : SUB$(R_i, L_j, L_k)$ - If the value stored in register $R_i$ is not zero, then subtract 1 from $R_i$ and jump to the instruction with label $L_j$, otherwise jump to the instruction with label $L_k$.

$L_h$ : HALT - the machine stops execution.

Starting from a register machine $M(m, H, L_0, L_h, I)$ we will construct a multiset reaction system $\mathcal{A} = (S, A)$ that simulates the behavior of $M$. The background set $S$ will contain elements modelling the state of the registers, the instruction labels, and auxiliary elements needed for the simulation. The set of reactions $A$ will include reactions that form each type of instruction in the register machine. We associate to each register in $M$ an element in $S$ such as the number of copies of that element in a state corresponds to the value stored in the register. All registers, denoted by $(0, 1, 2, ..., m-1)$, will have a corresponding element $R_r$ such that $r = 0, 1, ..., m-1$. If the value stored in register $R_r$ is $k$, then the state of the multiset reaction system will contain $k$ copies of the element $R_r$. The output register is register 0, therefore the number computed by the machine will be the number of copies of $R_0$ in the state when the machine halts. We also associate to each instruction label $L_i$ from $M$ an element $L_i$ in $S$ such that the presence of that element in a state will trigger the equivalent set of reactions for that label to be executed.

Reactions enabled within an instruction will be noted as $a_{ij}$ (for ADD instructions) and $s_{ij}$ (for SUB instructions) where $i$ corresponds to the instruction label and $j$ is an index to distinguish between different reactions within the same instruction. For the halt instruction we will simply note the specific reactions used in the implementation with $h_i$, where $i$ is used to define between different reactions.

Finally, we introduce auxiliary elements in $S$ to help with the simulation of the instructions. Such elements will be noted as $Z_{ij}$ where $i$ coresponds to the instruction label and $j$ is an index to distinguish between different auxiliary elements for the same instruction. We will note the auxiliary elements used in the implementation of the halt instruction with $H_i$ where $i$ is used to define between different reactants.

We propose a setting that will be able to do the simulation of the register machines: imposing sequentiality for the execution of reactions via a priority rule in the rection systems.

This approach was inspired by the max-sequentiality spiking of neural P systems [8] . The *max-sequentiality* condition for spiking neural P Systems described in [6] states that at every step, the systems chooses only one neuron that can fire from all active neurons, based on the maximum number of spikes stored. Similarly our multiset reaction system will execute only one reaction at the time and the reaction to be executed will be chosen from all currently enabled reactions by the maximum number of reactants.

The sequential SNP system encodes the number $n$ stored within a register as $2n + 3$ spikes within a coresponding neuron. We will use a similar encoding for the number stored in a register. Therefore, if the number $n$ is stored in register $R_r$, then the state of the multiset reaction system will contain $2n + 1$ copies of the element $R_r$.

Using these rules, we will now define the ADD, SUB and HALT instructions for small universal machines:

### 4.1 The ADD Instruction

We define the ADD instruction as follows: $L_i$: ADD$(R_r, L_j, L_k)$. This instructions adds 1 to the value stored in register $R_r$, then jumps non-deterministically to either the instruction with label $L_j$ or to the instruction with label $L_k$. For this instruction we define the following background set of elements and reactions:

$$\begin{aligned} S_{ADD} &= \{L_i, L_j, L_k, R_r\} \\ A_{ADD} &= \{a_{i1}, a_{i2}\} \\ a_{i1} &= (\{L_i\}, \{\dagger\}, \{R_r, R_r, L_j\}), \\ a_{i2} &= (\{L_i\}, \{\dagger\}, \{R_r, R_r, L_k\}) \end{aligned}$$

The presence of $L_i$ in a state will trigger the execution of the ADD instruction, which will produce two copies of the element $R_r$ and either $L_j$ or $L_k$ depending on the reaction that is executed. Generating two copies of $R_r$ effectively adds 1 to the value stored in register $R_r$ as we are using the encoding where the number $n$ is represented by $2n+1$ copies of the element $R_r$. In multiset reaction systems, each resource is available in a specific quantity, introducing a constraint on the execution of reactions. Both reactions have the same number of reactants, but only one of the two reactions enabled by the presence of $L_i$ can be executed, so the choice of which reaction to execute is non-deterministic. The execution flow of the reactions can be visualized in Figure 1.

### 4.2 The SUB Instruction

The SUB instruction is defined as follows: $L_i$: SUB$(R_r, L_j, L_k)$. This instruction checks if the value stored in register $R_r$ is not zero, then subtracts 1 from $R_r$ and jumps to the instruction with label $L_j$. Otherwise the value in $R_r$ remains zero and the program jumps to the instruction with label $L_k$. For this instruction we

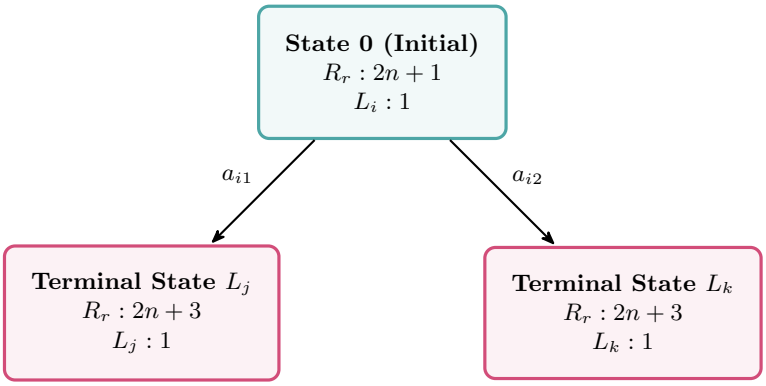



**Fig. 1.** The ADD module with priority-based sequential reaction execution.

define the following background set of elements and reactions:

$$
\begin{aligned}
S_{SUB} &= \{L_i, L_j, L_k, R_r, Z_{i1}, Z_{i2}, Z_{i3}, Z_{i4}\} \\
A_{SUB} &= \{s_{i1}, s_{i2}, s_{i3}, s_{i4}, s_{i5}, s_{i6}, s_{i7}\} \\
s_{i1} &= (\{L_i\}, \{\dagger\}, \{Z_{i1}, Z_{i1}, Z_{i3}\}), \\
s_{i2} &= (\{R_r, R_r, R_r, Z_{i3}\}, \{\dagger\}, \{Z_{i1}, Z_{i2}, R_r\}) \\
s_{i3} &= (\{Z_{i1}, Z_{i1}\}, \{\dagger\}, \{R_r, R_r\}), \\
s_{i4} &= (\{Z_{i1}, Z_{i1}, Z_{i1}\}, \{\dagger\}, \{Z_{i2}, Z_{i4}\}), \\
s_{i5} &= (\{Z_{i2}, Z_{i2}\}, \{\dagger\}, \emptyset), \\
s_{i6} &= (\{Z_{i4}\}, \{\dagger\}, \{L_j\}), \\
s_{i7} &= (\{Z_{i1}, Z_{i2}\}, \{\dagger\}, \{L_k\})
\end{aligned}
$$

The execution flow of the SUB instruction can be examined in Figure 2. The first reaction enabled is $s_{i1}$ as it is the only reaction enabled by the presence of $L_i$ in the state. The activation of $s_{i1}$ produces two copies of $Z_{i1}$ and one copy of $Z_{i3}$, which enables the reactions $s_{i2}$ and $s_{i3}$. Reactions $s_{i2}$ and $s_{i3}$ have 4 and 2 reactants respectively, so $s_{i2}$ is the reaction that will be executed as it has the highest number of reactants. However, reaction $s_{i2}$ can only be executed if there are at least 3 copies of $R_r$ in the state, which corresponds to the value stored in register $R_r$ being at least 1. This is where the execution of the reactions chain branches based on the value stored in $R_r$.

First, we will examine the case where the value stored in $R_r$ is at least 1. In this case, $s_{i2}$ is enabled and will be executed first, producing: a copy of the element $Z_{i1}$, a copy of the element $Z_{i2}$, and a copy of the element $R_r$; and consumes 3 copies of $R_r$ and one copy of $Z_{i3}$. After this state transition, the number of copies of $R_r$ is reduced by 2, which corresponds to the subtraction of 1 from the value stored in register $R_r$. Additionally, there are no more copies of $Z_{i3}$ in the state, which means that $s_{i2}$ is no longer enabled, and the only enabled reactions are $s_{i3}$, $s_{i4}$, and $s_{i7}$. Reaction $s_{i4}$ has the highest number of

reactants, so it will be executed next, producing a copy of $Z_{i2}$, a copy of $Z_{i4}$, and consuming 3 copies of $Z_{i1}$. After executing $s_{i4}$, the only enabled reactions are $s_{i5}$ and $s_{i6}$, which require 2 and 1 reactants respectively. Naturally $s_{i5}$ will be activated next, consuming 2 copies of $Z_{i2}$ and producing nothing, which leaves $s_{i6}$ as the only enabled reaction. The execution of $s_{i6}$ produces a copy of $L_j$, which corresponds to the jump to instruction $L_j$ in the register machine.

Now we will examine the case where the value stored in $R_r$ is zero. In this case, the number of copies of the element $R_r$ is 1, so reaction $s_{i2}$ is not enabled and that leaves $s_{i3}$ as the only reaction that can be activated. After the execution of $s_{i3}$, there are no more copies of $Z_{i1}$ in the state, which means that $s_{i3}$ cannot be enabled again. Additionally, two more copies of $R_r$ are produced, which means that $s_{i2}$ is now enabled. The execution of $s_{i2}$ produces a copy of $Z_{i1}$, a copy of $Z_{i2}$, and returns the number of copies of $R_r$ to their original count. The only enabled reaction now is $s_{i7}$, which produces a copy of $L_k$, corresponding to the jump to instruction $L_k$ in the register machine.

### 4.3 The HALT Instruction

The HALT instruction is defined as $L_h$: HALT. When $L_h$ is fired we need to compute the exact number $n$ stored in the register $R_0$ and then produce a copy of the element † to stop all possible reactions. To accomplish this, we define the following background set of elements and reactions:

$$\begin{aligned} S_{HALT} &= \{L_h, R_0, H_1, H_2\} \\ A_{HALT} &= \{h_1, h_2, h_3, h_4\} \\ h_1 &= (\{L_h, R_0, R_0\}, \{\dagger\}, \{L_h, H_1\}), \\ h_2 &= (\{L_h, R_0\}, \{\dagger\}, \{H_2\}) \\ h_3 &= (\{H_1, H_2\}, \{\dagger\}, \{R_0, H_2\}), \\ h_4 &= (\{H_2\}, \{\dagger\}, \{\dagger\}), \end{aligned}$$

Before the execution of the HALT instruction, the state contains $2n+1$ copies of $R_0$. The existence of $L_h$ in the state will enable the execution of reactions $h_1$ and $h_2$. If $n \geq 1$, then $h_1$ is the reaction with the highest number of reactants, so it will be executed first, producing a copy of $H_1$ and consuming two copies of $R_0$. As long as there are enough copies of $R_0$ in the state, we can keep executing reaction $h_1$, effectively producing $n$ copies of $H_1$ and consuming $2n$ copies of $R_0$. This leaves us with only one copy of $R_0$ and one copy of $L_h$ in the state. Now the only enabled reaction is $h_2$, which produces a copy of $H_2$ and consumes the last copy of $R_0$ and the last copy of $L_h$. In this next state, both $h_3$ and $h_4$ are enabled, but $h_3$ has more reactants, so it will be executed first. We will now proceed to execute reaction $h_3$ and restore the value $n$ in the register $R_0$. Activating $h_3$ produces a copy of $R_0$ and consumes one copy of $H_1$ and one copy of $H_2$. The element $H_2$ is both consumed and produced by $h_3$, ensuring that all reactions happen sequentially. When there are no more copies of $H_1$ in the state,

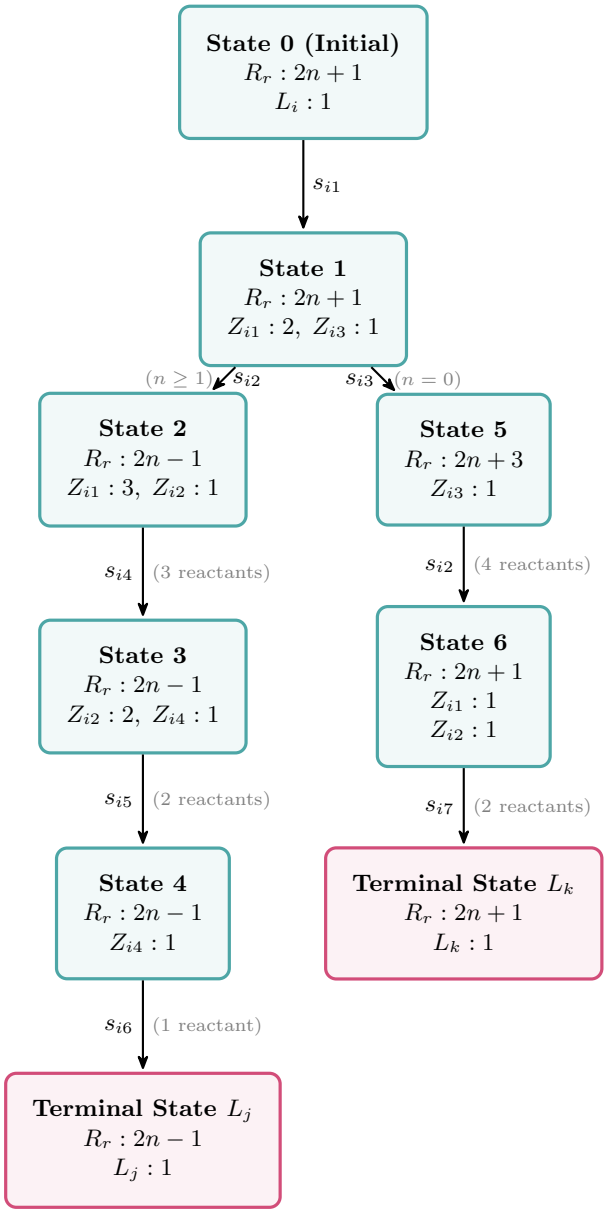



**Fig. 2.** The SUB module with priority-based sequential reaction execution.

the only enabled reaction is $h_4$, which produces a copy of the universal inhibitor and consumes the last copy of $H_2$, effectively halting the system.

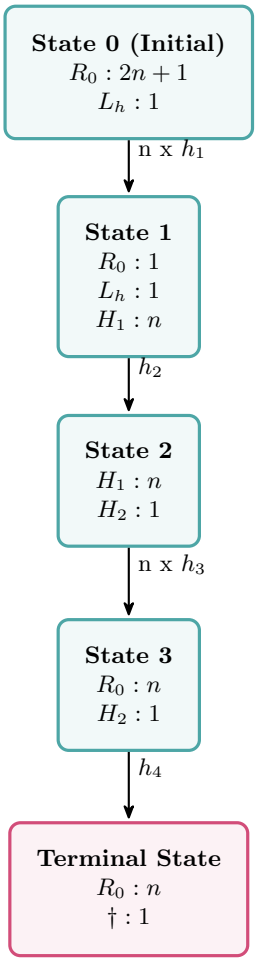



**Fig. 3.** The HALT module with priority-based sequential reactions execution.

## 5 Conclusions

We introduced the implementation of a small universal machine of 23 instructions using multiset reactions systems. We encoded the registers and instructions labels as specific elements. We have shown that all types of instructions can be modeled using reaction chains. Reactions chains that model a specific instruction can only be enabled once the element encoding the instruction label is generated, ensuring that instructions are executed sequentially.

The approach considers that reactions are executed sequentially based on a priority order defined by the number of reactants (max sequentiality). This means that for all enabled reactions in a state, the reaction with the most num-

ber of reactants will be chosen to activate. We have concluded that 87 distinct elements are needed to build the small universal machine using this approach.

Further research could consider the use of other approaches to enforce sequentiality, namely the variation in execution time or using inhibitors to control the flow of reactions. Moreover, the number of elements used could be reduced by optimizing the number of auxiliary elements needed to build the reaction chains, or by implementing a different small universal machine model with fewer instructions.

**Acknowledgments.** This study was partially supported by Core Program within the National Research, Development and Innovation Plan 2022-2027, carried out with the support of ANC, project no. 23020101(SIA-PRO), contract no 7N/2022.

## References

1. Ehrenfeucht, A., & Rozenberg, G. (2004, December). Basic notions of reaction systems. In International Conference on Developments in Language Theory (pp. 27-29). Berlin, Heidelberg: Springer Berlin Heidelberg.
2. Ehrenfeucht, A., & Rozenberg, G. (2007). Reaction systems. Fundamenta informaticae, 75(1-4), 263-280.
3. Bottoni, P., Mitrana, V., Petre, I. (2025). Multiset Reaction Systems. In: Jiménez López, M.D., Vaszil, G. (eds) Languages of Cooperation and Communication. Lecture Notes in Computer Science, vol 15840
4. Korec, I. (1996). Small universal register machines. Theoretical Computer Science, 168(2), 267-301.
5. Păun, A., & Păun, G. (2007). Small universal spiking neural P systems. BioSystems, 90(1), 48-60.
6. Ibarra, O. H., Păun, A., & Rodríguez-Patón, A. (2009). Sequential SNP systems based on min/max spike number. Theoretical Computer Science, 410(30-32), 2982-2991.
7. Minsky, M. L. (1967). Computation. Englewood Cliffs: Prentice-Hall.
8. Ionescu, M., Păun, G., & Yokomori, T. (2006). Spiking neural P systems. Fundamenta informaticae, 71(2-3), 279-308.

# Algebraic Topological Data Analysis of Membrane Systems via Multi-Hypergraph Incidence Models

Pedro Real

Instituto de Ingeniería Informática, I3US
University of Seville, Spain
real@us.es

**Abstract.** We introduce an algebraic-topological framework for membrane systems based on multi-hypergraph incidence models. A membrane computation step is encoded through six incidence-type matrices capturing structural support, rule localization, enabling conditions, rule interaction, control policies and semantic effects. These layers are assembled into a lower block-triangular global incidence matrix, whose active part is selected configuration-dependently. Each incidence operator is interpreted as a one-dimensional chain complex, and we associate to each a homological twin (ATWIN) comprising an explicit AT-model, Betti numbers, torsion data (where applicable), generalized inverses, projectors and homology representatives. From these ATWINs we extract ATDA (Algebraic Topological Data Analysis) descriptors for component topology, activation patterns, semantic aggregation and, when the full global matrix is assembled, inter-layer coupling. Semantic aggregation is treated as descriptor comparison between fine-grained and attributed models.
Two complementary proof-of-concept experiments illustrate the framework at distinct hierarchy levels. A minimal channel-state tissue P system illustrates component-level topology through structural and control cycles. The Basic Counter and Decision (BCD) transition P system, interpreted over the MAREX architectural support, illustrates configuration-dependent active semantic topology: ATDA descriptors separate the potential topology of the rule system from the topology active during each computation phase, detect the transition from parallel counting to priority-controlled decision, and distinguish the negative trajectory producing $Q + \delta$ from the positive trajectory producing $P + \delta$. Finally, we outline an extension toward time-scale topological dynamical systems via the Topological Scale Framework.

**Keywords:** Membrane computing · Hypergraphs · Incidence matrices · Algebraic topology · AT-models · Smith Normal Form · Topological Data Analysis · MAREX · Topological Scale Framework

## 1 Introduction

Membrane computing was introduced by Păun as a computational model inspired by the compartmental organization and distributed dynamics of biologi-

cal membranes [1,2]. In its classical formulation, a membrane system consists of a hierarchical membrane structure, multisets of objects placed in regions, and evolution or communication rules applied in a maximally parallel way. This formalism has proved successful in computability, complexity and modelling. However, the rooted membrane tree mainly records inclusion, whereas communication channels, rule dependencies, synchronization constraints, hardware supports and higher-order interactions are often encoded only indirectly.

This motivates an incidence-based view of membrane systems, especially for network-oriented, implementation-oriented and simulation-oriented variants such as tissue P systems, channel-state systems, spiking neural P systems, MAREX-like architectures and multi-threaded simulators. In these settings, a computation step depends on explicit incidences among objects, rules, regions, control units and communication supports: rules are localized, resources enable or disable them, rules may interact, control policies select admissible actions, and semantic effects update object multiplicities.

We propose an algebraic-topological representation based on multi-hypergraph incidence models. A membrane program is decomposed into six operational layers: structural support, localization, enabling, rule interaction, control and semantics. These layers are assembled into a lower block-triangular global incidence matrix $M_{\mathrm{glob}}$, whose diagonal blocks describe the internal incidence of each layer and whose off-diagonal blocks encode feed-forward coupling. A configuration-dependent selector $C_{\mathrm{glob}}(q_t)$ defines the active matrix

$$M_{\mathrm{glob}}^{\mathrm{act}}(q_t) = M_{\mathrm{glob}} C_{\mathrm{glob}}(q_t),$$

which represents the incidence structure actually used during the transition from $q_t$ to $q_{t+1}$.

Every incidence-type matrix $T$ in the model is then interpreted as a one-dimensional chain complex $0 \longrightarrow \mathbb{F}[E] \xrightarrow{T} \mathbb{F}[V] \longrightarrow 0$. This applies to component matrices, transition blocks, global and active matrices, and semantic aggregations. For each operator we construct a homological twin, or ATWIN, using AT-model reductions and Smith Normal Form. The associated ATDA descriptor records

$$(n_T, m_T, r_T, \beta_0(T), \beta_1(T), \chi(T), \mathrm{Tor}\, H_0(T)).$$

These descriptors provide computable summaries of component topology, active topology, semantic aggregation and, when the full global matrix is assembled, inter-layer coupling.

The main contributions are:

- a lower block-triangular multi-hypergraph incidence model for one membrane computation step;
- configuration-dependent active incidence matrices for selected transitions;
- a hierarchy of ATWIN models and ATDA descriptors covering component, transition, global, active and semantic incidence operators;
- a semantic aggregation procedure comparing fine and attributed incidence models through their ATWINs and ATDA descriptors;

– two focused proof-of-concept examples: a minimal channel-state tissue P system illustrating component topology, and a Basic Counter and Decision (BCD) transition P system [17], interpreted over the MAREX architectural support [21], showing how active ATDA descriptors distinguish parallel counting, priority-controlled decision and dissolution phases.

The rest of the paper is organized as follows. Section 3 introduces the multi-hypergraph incidence model. Section 4 constructs ATWINs for incidence operators. Section 5 defines the ATDA descriptor hierarchy. Section 6 presents two proof-of-concept experiments: a minimal channel-state tissue P system and the BCD–MAREX active semantic case study. Section 7 concludes and outlines future work.

## 2 Related Work

Membrane computing originates in Păun's model of distributed computation inspired by biological compartmentalization [1,2]. Many variants have since been introduced, including tissue P systems, symport/antiport systems, active membrane systems, spiking neural P systems and systems with dynamically evolving membrane structures [23,20,24,22,11]. These models make communication, synchronization, competition and interaction central to the computation. Tissue and channel-state variants are especially close to an incidence viewpoint, because their dynamics depends explicitly on relations among cells, objects, channels and rules [12]. Topological approaches to membrane computing have also been developed: Giavitto and Michel introduced topological collections and local transformations [6,7], and Dennunzio et al. studied topological properties of configuration spaces for deterministic P systems with active membranes [5].

Hypergraphs are a natural language for higher-order interactions, since one hyperedge may relate several entities at once. They have been used in communication-oriented P systems, hypergraph-based spiking neural P systems and membrane-inspired learning architectures [18,16,19,8]. The present work follows this incidence perspective, but uses a unified multi-layer model in which component matrices, transition blocks, global matrices and active matrices are analysed by the same homological machinery. The framework is also related to Petri nets, where state evolution is described by matrices encoding consumption and production relations [30].

The experimental setting is connected with recent implementation work in membrane computing. MAREX provides a hardware architecture based on a Global Object Container, an Object Circulating Bus, Rule Container Units and a Membrane Control Unit [21]. The BCD system used here comes from a multi-threaded simulation approach in which object types and rules are coordinated through synchronized threads [17]. The algebraic-topological part relies on standard algebraic topology, computational homology, AT-models and chain contractions [9,10,26,27]. The aim is not to replace the operational semantics of P systems or to provide a faster simulator, but to extract structural, active and semantic topological descriptors of membrane programs and selected computation steps.

# 3 A Multi-Hypergraph Incidence Model for Membrane Computation

We start from a membrane system $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m)$, where $O$ is the object alphabet, $\mu$ is the membrane structure, $w_i$ is the initial multiset in region $i$, and $R_i$ is the set of rules associated with that region [1,2]. A computation generates a sequence $q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow \cdots$. We replace the direct multiset-rewriting description of one transition by an incidence-based algebraic model: the membrane program is decomposed into layers, assembled into a global incidence matrix, and activated according to the current configuration.

*Operational scope.* In this paper we do not fix a single maximally specific variant of P systems. Instead, we consider one computation step of a membrane system whose operational semantics determines, from the current configuration $q_t$, a set of applicable and selected rule occurrences. This covers standard evolution-rule systems as well as tissue, channel-state and implementation-oriented variants, provided that applicability, control and semantic effects can be represented by finite incidence data. Thus the proposed construction starts after the usual operational semantics has determined the active rule incidences for the current step; it does not replace that semantics.

## 3.1 Hypergraph Layers

We encode the operational information of a membrane system by six hypergraph layers $\mathcal{H}_s, \mathcal{H}_a, \mathcal{H}_B, \mathcal{H}_r, \mathcal{H}_\chi, \mathcal{H}_\Sigma$. Each layer is $\mathcal{H}_X = (V_X, E_X, M_X)$, with $X \in \{s, a, B, r, \chi, \Sigma\}$, where $M_X : \mathbb{F}[E_X] \longrightarrow \mathbb{F}[V_X]$, $M_X \in \mathbb{F}^{|V_X| \times |E_X|}$, is its vertex–hyperedge incidence matrix. The six layers respectively encode structural support, rule localization, enabling conditions, rule interaction, control and semantic production–consumption effects; see Table 1.

**Table 1.** Hypergraph layers used to represent a membrane system.

| Layer | Hypergraph | Encoded information | Role in one step |
|---|---|---|---|
| Structural | $\mathcal{H}_s$ | Membranes, regions, channels, adjacency | Computational support |
| Localization | $\mathcal{H}_a$ | Rule positions and supports | Places rules |
| Enabling | $\mathcal{H}_B$ | Objects, resources, applicability | Selects candidates |
| Rule interaction | $\mathcal{H}_r$ | Dependencies, conflicts, compatibilities | Organizes enabled rules |
| Control | $\mathcal{H}_\chi$ | Priorities, inhibitors, synchronization | Selects admissible rules |
| Semantics | $\mathcal{H}_\Sigma$ | Production–consumption effects | Produces $\Delta q_t$ |

The labels have the following mnemonic meaning: $s$ denotes structural support, $a$ denotes rule allocation or localization, $B$ denotes Boolean enabling conditions, $r$ denotes rule–rule interaction, $\chi$ denotes control or choice, and $\Sigma$ denotes semantic production–consumption effects.

The order $s \longrightarrow a \longrightarrow B \longrightarrow r \longrightarrow \chi \longrightarrow \Sigma$ is not an extra restriction on the membrane system. It only records the causal processing order of one transition: support, localization, enabling, interaction, control and semantic update. Feedback is encoded by temporal iteration, since the output configuration $q_{t+1}$ becomes the input of the next step.

### 3.2 Global Operational Incidence Matrix

The six layers are assembled into a global hypergraph $\mathcal{H}_{\text{glob}} = (V_{\text{glob}}, E_{\text{glob}}, M_{\text{glob}})$, with $\mathbb{F}[V_{\text{glob}}] = \bigoplus_X \mathbb{F}[V_X]$ and $\mathbb{F}[E_{\text{glob}}] = \bigoplus_X \mathbb{F}[E_X]$. Let $n_X = |V_X|$, $m_X = |E_X|$, $n_{\text{glob}} = \sum_X n_X$ and $m_{\text{glob}} = \sum_X m_X$. The global incidence matrix $M_{\text{glob}} : \mathbb{F}[E_{\text{glob}}] \longrightarrow \mathbb{F}[V_{\text{glob}}]$ has diagonal blocks $M_X$ and off-diagonal coupling blocks $K_{YX} : \mathbb{F}[E_X] \longrightarrow \mathbb{F}[V_Y]$ between consecutive layers. With the order $(s, a, B, r, \chi, \Sigma)$,

$$M_{\text{glob}} = \begin{pmatrix} M_s & 0 & 0 & 0 & 0 & 0 \\ K_{as} & M_a & 0 & 0 & 0 & 0 \\ 0 & K_{Ba} & M_B & 0 & 0 & 0 \\ 0 & 0 & K_{rB} & M_r & 0 & 0 \\ 0 & 0 & 0 & K_{\chi r} & M_\chi & 0 \\ 0 & 0 & 0 & 0 & K_{\Sigma\chi} & M_\Sigma \end{pmatrix}.$$

The blocks $K_{as}$, $K_{Ba}$, $K_{rB}$, $K_{\chi r}$ and $K_{\Sigma\chi}$ transfer information from structure to localization, from localization to enabling, from enabling to interaction, from interaction to control, and from control to semantics. The lower block-triangular form represents one feed-forward computation step.

### 3.3 Configuration-Dependent Activation

The matrix $M_{\text{glob}}$ describes all potential incidences. At configuration $q_t$, only part of them is active. We introduce a selector $C_{\text{glob}}(q_t) : \mathbb{F}[E_{\text{glob}}] \longrightarrow \mathbb{F}[E_{\text{glob}}]$, $C_{\text{glob}}(q_t) \in \mathbb{F}^{m_{\text{glob}} \times m_{\text{glob}}}$. A typical form is

$$C_{\text{glob}}(q_t) = \operatorname{diag}\big(I_{E_s}, I_{E_a}, C_B(q_t), I_{E_r}, C_\chi(q_t), C_\Sigma(q_t)\big).$$

Here $C_B(q_t)$ selects satisfied enabling incidences, $C_\chi(q_t)$ selects rules admitted by the control policy, and $C_\Sigma(q_t)$ selects the realized semantic effects. In the Boolean case, $C_B(q_t) = \operatorname{diag}(c_e(q_t))_{e \in E_B}$, where

$$c_e(q_t) = \begin{cases} 1, & \text{if the condition represented by } e \text{ is satisfied in } q_t, \\ 0, & \text{otherwise.} \end{cases}$$

The active global incidence matrix is $M^{\text{act}}_{\text{glob}}(q_t) = M_{\text{glob}} C_{\text{glob}}(q_t)$. Thus $M_{\text{glob}}$ is the potential architecture, while $M^{\text{act}}_{\text{glob}}(q_t)$ is the incidence structure used in the current transition.

### 3.4 Computation as a Global Incidence Transfer

Let $Q$ be the set of elementary configuration coordinates, and let $q_t \in \mathbb{F}[Q]$. Since $M_{\mathrm{glob}}$ acts on $\mathbb{F}[V_{\mathrm{glob}}]$, we use an insertion map $J_Q : \mathbb{F}[Q] \longrightarrow \mathbb{F}[V_{\mathrm{glob}}]$ and a realization map $S_Q : \mathbb{F}[V_{\mathrm{glob}}] \longrightarrow \mathbb{F}[Q]$. The selected active hyperedge vector is

$$u_t = C_{\mathrm{glob}}(q_t) M_{\mathrm{glob}}^T J_Q q_t.$$

The induced configuration variation is

$$\Delta q_t = S_Q M_{\mathrm{glob}} C_{\mathrm{glob}}(q_t) M_{\mathrm{glob}}^T J_Q q_t.$$

Hence the selected incidence-mediated transition is

$$q_{t+1} = \Psi_{q_t}(q_t), \qquad \Psi_{q_t} = I_Q + S_Q M_{\mathrm{glob}} C_{\mathrm{glob}}(q_t) M_{\mathrm{glob}}^T J_Q.$$

This formula records one selected incidence-mediated transition after the operational semantics has fixed $C_{\mathrm{glob}}(q_t)$. It is not a replacement for the full operational semantics of membrane-system variants. Once $C_{\mathrm{glob}}(q_t)$ is fixed, the transfer is linear; the complete dynamics is generally nonlinear because selection depends on the current configuration and encodes applicability, competition, inhibition, synchronization and maximal-parallel policies.

### 3.5 Semantic Aggregation

The word "semantic" is used here in an incidence-descriptor sense. It does not denote a new operational semantics for P systems. The operational semantics still determines which rules are applicable and selected. The semantic layer records the production–consumption effects of those rules, and semantic aggregation groups vertices or hyperedges according to attributes such as object type, rule family, phase or output role.

The same model admits semantic aggregation. Let $A_V^{\mathrm{glob}} : \mathbb{F}[V_{\mathrm{glob}}] \longrightarrow \mathbb{F}[\widetilde{V}_{\mathrm{glob}}]$ and $A_E^{\mathrm{glob}} : \mathbb{F}[E_{\mathrm{glob}}] \longrightarrow \mathbb{F}[\widetilde{E}_{\mathrm{glob}}]$ be vertex- and hyperedge-attribute aggregation maps. The semantic global incidence matrix is

$$\widetilde{M}_{\mathrm{glob}} = A_V^{\mathrm{glob}} M_{\mathrm{glob}} \big(A_E^{\mathrm{glob}}\big)^T.$$

Thus the model provides three related matrices: $M_{\mathrm{glob}}$, describing the potential incidence structure; $M_{\mathrm{glob}}^{\mathrm{act}}(q_t)$, describing the active structure at time $t$; and $\widetilde{M}_{\mathrm{glob}}$, describing the semantic aggregation.

## 4 Rational AT-Models and Homological ATWINs

We now attach to each incidence operator a computable algebraic-topological model, called an ATWIN. This applies to component matrices, transition blocks, attributed and semantic matrices, and global or active incidence matrices.

### 4.1 Incidence Operators as One-Dimensional Chain Complexes

Intuitively, $\beta_0$ counts independent connected components left by the incidence relations, while $\beta_1$ counts independent cycles or degrees of freedom among hyperedges. Over the integers, the Smith normal form also detects torsion in $H_0$, that is, finite cyclic obstructions not visible from ranks over a field.

Let $T : R[E] \longrightarrow R[V]$ be an incidence-type operator, with $R = \mathbb{F}$ or $R = \mathbb{Z}$. It defines the chain complex $C(T;R) : 0 \longrightarrow R[E] \xrightarrow{T} R[V] \longrightarrow 0$, so that $H_1(T;R) = \ker T$ and $H_0(T;R) = \operatorname{coker} T$. If $R = \mathbb{F}$ and $T \in \mathbb{F}^{n\times m}$ has rank $r = \operatorname{rank}_{\mathbb{F}}(T)$, then $\beta_0(T) = n - r$ and $\beta_1(T) = m - r$.

For integer matrices, torsion is obtained from the Smith normal form. If

$$UTV = D = \operatorname{diag}(d_1, \ldots, d_r, 0, \ldots, 0), \qquad d_i \mid d_{i+1},$$

then $H_0(T;\mathbb{Z}) \simeq \mathbb{Z}^{n-r} \oplus \bigoplus_{d_i>1} \mathbb{Z}_{d_i}$, while $H_1(T;\mathbb{Z})$ is free. Hence $\operatorname{Tor} H_0(T;\mathbb{Z}) \simeq \bigoplus_{d_i>1} \mathbb{Z}_{d_i}$. Thus Betti numbers are field invariants, whereas torsion is an integral invariant.

### 4.2 The Rational ATWIN of an Incidence Operator

Assume that $T$ is integer-valued. From the Smith normal form, define $D^{\phi}$ as the rectangular diagonal matrix of size $m \times n$ obtained from $D$ by transposition and by replacing each nonzero elementary divisor $d_i$ by $d_i^{-1}$. If $UTV = D$, set

$$T^{\phi} = V D^{\phi} U.$$

Then $TT^{\phi}T = T$ and $T^{\phi}TT^{\phi} = T^{\phi}$. The associated projectors are $P_0(T) = I_{C_0} - TT^{\phi}$ and $P_1(T) = I_{C_1} - T^{\phi}T$, whose images give representatives of $H_0(T)$ and $H_1(T)$, respectively. We set $H(T) = H_1(T) \oplus H_0(T)$ with $H_1(T) = \operatorname{im} P_1(T)$, $H_0(T) = \operatorname{im} P_0(T)$ and zero differential.

The rational AT-model is the contraction $(f_T, g_T, \phi_T) : C(T) \rightleftarrows H(T)$, where $f_T = P_1(T) \oplus P_0(T)$, $g_T : \operatorname{im} P_1(T) \oplus \operatorname{im} P_0(T) \hookrightarrow C_1(T) \oplus C_0(T)$ and $\phi_T|_{C_0} = T^{\phi}$ and $\phi_T|_{C_1} = 0$. It satisfies the following equalities: $f_T g_T = \mathrm{id}_{H(T)}$ and $\mathrm{id}_{C(T)} - g_T f_T = d\phi_T + \phi_T d$. We define

$$\mathrm{ATWIN}(T) = \big(H(T), f_T, g_T, \phi_T, P_0(T), P_1(T), T^{\phi}\big).$$

Thus ATWIN contains the homology, representatives, projectors and a rational homotopy inverse. For real or weighted operators one may use a generalized inverse over $\mathbb{R}$; Betti numbers and projectors remain meaningful, but integral torsion is not defined.

### 4.3 Families of ATWINs in the Multi-Hypergraph Model

The construction is applied uniformly to all incidence operators in the model:

$$\begin{array}{ll}
\mathrm{ATWIN}(M_X), & X \in \{s, a, B, r, \chi, \Sigma\}, \\
\mathrm{ATWIN}(K_{YX}), & X \longrightarrow Y, \\
\mathrm{ATWIN}(A_V^{(X)}),\ \mathrm{ATWIN}(A_E^{(X)}), & \text{attribute maps}, \\
\mathrm{ATWIN}(\widetilde{M}_X), & \widetilde{M}_X = A_V^{(X)} M_X \big(A_E^{(X)}\big)^T, \\
\mathrm{ATWIN}(M_{\mathrm{glob}}), & \text{global operational matrix}, \\
\mathrm{ATWIN}(M_{\mathrm{glob}}^{\mathrm{act}}(q_t)), & M_{\mathrm{glob}}^{\mathrm{act}}(q_t) = M_{\mathrm{glob}} C_{\mathrm{glob}}(q_t), \\
\mathrm{ATWIN}(\widetilde{M}_{\mathrm{glob}}), & \widetilde{M}_{\mathrm{glob}} = A_V^{\mathrm{glob}} M_{\mathrm{glob}} \big(A_E^{\mathrm{glob}}\big)^T.
\end{array}$$

These ATWINs describe, respectively, component topology, inter-layer coupling, attribution, semantic aggregation, global topology, active transition topology and semantic global topology. The numerical and algebraic descriptors used for comparison in the next sections are extracted from these ATWINs and are called ATDA descriptors.

## 5 ATDA Descriptors of the Multi-Hypergraph Incidence Model

ATDA descriptors are the numerical and algebraic features extracted from ATWINs. They are associated with all incidence-type matrices in the model:

$$M_X,\ K_{YX},\ M_{\mathrm{glob}},\ M_{\mathrm{glob}}^{\mathrm{act}}(q_t),\ A_V^{(X)},\ A_E^{(X)},\ \widetilde{M}_X,\ \widetilde{K}_{YX},\ \widetilde{M}_{\mathrm{glob}}.$$

### 5.1 Basic Descriptor

For an incidence operator $T : \mathbb{F}[E_T] \longrightarrow \mathbb{F}[V_T]$, let $n_T = |V_T|$, $m_T = |E_T|$ and $r_T = \mathrm{rank}(T)$. Since $C(T) : 0 \longrightarrow \mathbb{F}[E_T] \xrightarrow{T} \mathbb{F}[V_T] \longrightarrow 0$,

$$\beta_0(T) = n_T - r_T, \qquad \beta_1(T) = m_T - r_T, \qquad \chi(T) = n_T - m_T.$$

If $T$ is integer-valued, its Smith normal form gives $\mathrm{Tor}\, H_0(T) \simeq \bigoplus_{d_i>1} \mathbb{Z}_{d_i}$. We define $\mathrm{ATDA}(T) = (n_T,\, m_T,\, r_T,\, \beta_0(T),\, \beta_1(T),\, \chi(T),\, \mathrm{Tor}\, H_0(T))$. For non-integer or field-valued matrices, the torsion entry is omitted.

### 5.2 Descriptor Families

The basic descriptor is applied to the incidence operators introduced above. We use

$$\mathrm{ATDA}_{\mathrm{comp}}(\Pi) = \{\mathrm{ATDA}(M_X)\}_X, \qquad \mathrm{ATDA}_{\mathrm{tr}}(\Pi) = \{\mathrm{ATDA}(K_{YX})\}_{X \to Y},$$

for component and transition descriptors, and

$$\mathrm{ATDA}_{\mathrm{glob}}(\Pi) = \mathrm{ATDA}(M_{\mathrm{glob}}), \qquad \mathrm{ATDA}_{\mathrm{act}}(q_t) = \mathrm{ATDA}\big(M_{\mathrm{glob}}^{\mathrm{act}}(q_t)\big),$$

with

$$M^{\mathrm{act}}_{\mathrm{glob}}(q_t) = M_{\mathrm{glob}} C_{\mathrm{glob}}(q_t).$$

Along a trajectory $q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_T$, the active descriptors form

$$\mathrm{ATDA}_{\mathrm{traj}}(\Pi) = \left(\mathrm{ATDA}_{\mathrm{act}}(q_0), \ldots, \mathrm{ATDA}_{\mathrm{act}}(q_T)\right).$$

Semantic aggregation is treated as descriptor comparison rather than as a homology-preserving map. Thus, for

$$\widetilde{M}_X = A_V^{(X)} M_X \left(A_E^{(X)}\right)^T, \qquad \widetilde{M}_{\mathrm{glob}} = A_V^{\mathrm{glob}} M_{\mathrm{glob}} \left(A_E^{\mathrm{glob}}\right)^T,$$

we compute

$$\mathrm{ATDA}_{\mathrm{sem}}(\Pi) = \{\mathrm{ATDA}(\widetilde{M}_X)\}_X \cup \{\mathrm{ATDA}(\widetilde{M}_{\mathrm{glob}})\}.$$

The attribute maps $A_V^{(X)}$ and $A_E^{(X)}$ may also be analysed as incidence-type operators, but their descriptors measure the semantic grouping itself; they are not used here to define homology morphisms.

### 5.3 Comparative Descriptors

The effect of activation, coupling and semantic aggregation is measured by Betti differences:

$$\Delta_{\mathrm{act}}\beta_i(q_t) = \beta_i\left(M^{\mathrm{act}}_{\mathrm{glob}}(q_t)\right) - \beta_i(M_{\mathrm{glob}}), \qquad i = 0, 1,$$

$$\Delta_{\mathrm{coup}}\beta_i = \beta_i(M_{\mathrm{glob}}) - \sum_X \beta_i(M_X), \qquad i = 0, 1,$$

$$\Delta_{\mathrm{sem}}\beta_i = \beta_i(\widetilde{M}_{\mathrm{glob}}) - \beta_i(M_{\mathrm{glob}}), \qquad i = 0, 1,$$

and locally by

$$\Delta_{\mathrm{sem}}\beta_i(X) = \beta_i(\widetilde{M}_X) - \beta_i(M_X).$$

These quantities provide a comparison vocabulary for potential, active, coupled and semantically aggregated incidence topologies. In the present conference version we use only the component descriptors and the active semantic descriptors; full global, coupling and semantic-aggregation comparisons are left for an extended implementation study.

## 6 Experimental Case Studies

All descriptors reported below are obtained directly from the displayed incidence encodings. Ranks and Betti numbers are computed over $\mathbb{Q}$, while torsion entries are obtained from the Smith normal form over $\mathbb{Z}$. In the BCD example, active semantic matrices are obtained by restricting the potential semantic matrix to the explicitly listed active rule columns.

### 6.1 A Minimal Channel-State Example: Component Topology

We first include a minimal channel-state tissue P system only to illustrate the component level of the ATDA hierarchy. It is not intended as a benchmark, but as a minimal diagnostic example showing that structural and control incidence layers may already carry nontrivial Betti numbers. The system has three cells and the environment, $C = \{1, 2, 3, 0\}$, connected by the communication channels $c_{12}, c_{23}, c_{30}$ and $c_{01}$. We consider

$$r_1 : (a, 1) \xrightarrow{c_{12},\alpha} (b, 2), \qquad r_2 : (b, 2) \xrightarrow{c_{23},\beta} (c, 3), \qquad r_3 : (c, 3) \xrightarrow{c_{30},\gamma} (d, 0),$$

together with the cyclic channel-state control

$$\alpha \longrightarrow \beta \longrightarrow \gamma \longrightarrow \alpha.$$

The structural layer $M_s$ is the unsigned incidence matrix of the communication cycle on the vertices $1, 2, 3, 0$. The localization layer $M_a$ assigns the three rules to their channels. The enabling layer $M_B$ records the dependence of each rule on its required object and channel state. The control layer $M_\chi$ is the unsigned incidence matrix of the cycle $\alpha \to \beta \to \gamma \to \alpha$. The semantic layer $M_\Sigma$ is the signed production–consumption matrix of the three rules.

The ranks in Table 2 are computed from these incidence encodings; the example is not intended to model all channel-state tissue P systems, but to show how cycles in the communication support and in the cyclic control component are detected by the descriptor.

**Table 2.** Component ATDA descriptors for the minimal channel-state example.

| Layer | $n$ | $m$ | $r$ | $\beta_0$ | $\beta_1$ | $\chi$ | Meaning |
|---|---|---|---|---|---|---|---|
| Structural $M_s$ | 4 | 4 | 3 | 1 | 1 | 0 | communication cycle |
| Localization $M_a$ | 3 | 3 | 3 | 0 | 0 | 0 | rule–channel assignment |
| Enabling $M_B$ | 6 | 3 | 3 | 3 | 0 | 3 | applicability tests |
| Control $M_\chi$ | 3 | 3 | 2 | 1 | 1 | 0 | cyclic channel-state control |
| Semantics $M_\Sigma$ | 8 | 3 | 3 | 5 | 0 | 5 | independent rule effects |

The descriptor detects two cycles: $\beta_1(M_s) = 1$ and $\beta_1(M_\chi) = 1$. The first is structural, corresponding to the closed communication support among cells and environment. The second is operational, corresponding to the cyclic evolution of channel states. In contrast, the localization, enabling and semantic layers have $\beta_1 = 0$, so they behave as independent assignments or effects in this minimal example. Thus the channel-state system provides a component-level diagnostic: ATDA detects cycles already present in the structural and control incidence layers, independently of the active semantic analysis used in the following BCD–MAREX case.

### 6.2 BCD–MAREX: Active Semantic ATDA Trajectories

We consider the *Basic Counter and Decision* (BCD) transition P system introduced in [17], interpreted over the MAREX architectural support [21]. The

example is small and is used as an active-semantic diagnostic: it combines parallel counting, phase control, priority-based decision, output selection and membrane dissolution. MAREX supplies the structural support, while the BCD rules define the membrane program.

The system counts active neighbouring objects and produces $P$ if the number is greater than 3, and $Q$ otherwise. We use $N = 8$, with alphabet $\Gamma = \{E_i : 0 \leq i \leq 7\} \cup \{A, P, Q, F_1, F_2, \delta\}$. Here $E_i$ are active neighbours, $A$ is the counter, $F_1, F_2$ are phase markers, $P, Q$ are outputs and $\delta$ denotes dissolution. The rules are

$$R_{1,i} : E_i \longrightarrow A, \quad 0 \leq i \leq 7, \qquad R_2 : F_1 \longrightarrow F_2,$$
$$R_{3,i} : A^i F_2 \longrightarrow P, \delta, \quad 4 \leq i \leq 8, \qquad R_{4,j} : A^j F_2 \longrightarrow Q, \delta, \quad 1 \leq j \leq 3,$$
$$R_5 : F_2 \longrightarrow Q, \delta.$$

The decision rules are ordered by the priority chain

$$R_{3,8} > R_{3,7} > R_{3,6} > R_{3,5} > R_{3,4} > R_{4,3} > R_{4,2} > R_{4,1} > R_5.$$

Thus, after counting, the highest applicable decision rule selects the final output.

*Configurations and incidence encoding.* We compare two trajectories with the same potential rule system but different active incidences. The negative configuration has three active neighbours, $q_0^- = F_1 + E_1 + E_3 + E_7$, and evolves as $q_0^- \longrightarrow q_1^- = 3A + F_2 \longrightarrow q_2^- = Q + \delta$. The positive configuration has four active neighbours, $q_0^+ = F_1 + E_1 + E_3 + E_5 + E_7$, and evolves as $q_0^+ \longrightarrow q_1^+ = 4A + F_2 \longrightarrow q_2^+ = P + \delta$. In the first counting phase the active rules are $\{R_{1,1}, R_{1,3}, R_{1,7}, R_2\}$ and $\{R_{1,1}, R_{1,3}, R_{1,5}, R_{1,7}, R_2\}$, respectively. In the decision phase the selected rules are $R_{4,3}$ and $R_{3,4}$.

The BCD system is encoded through the six-layer pipeline

$$s \longrightarrow a \longrightarrow B \longrightarrow r \longrightarrow \chi \longrightarrow \Sigma.$$

The structural layer $M_s$ is the MAREX support with vertices

$$V_s = (\mathrm{GOC}, \mathrm{OCB}, \mathrm{RCU}, \mathrm{MCU}, \mathrm{Thread}, \mathrm{Barrier}).$$

The localization layer is $M_a = I_{18}$. The enabling layer is the consumption matrix $M_B = C^-$, with rows $(E_0, \ldots, E_7, A, F_1, F_2, P, Q, \delta)$ and columns

$$(R_{1,0}, \ldots, R_{1,7}, R_2, R_{3,4}, \ldots, R_{3,8}, R_{4,1}, R_{4,2}, R_{4,3}, R_5).$$

Its nonzero entries are

$$C^-_{E_i,R_{1,i}} = 1, \quad C^-_{F_1,R_2} = 1, \quad C^-_{A,R_{3,i}} = i, \quad C^-_{F_2,R_{3,i}} = 1,$$
$$C^-_{A,R_{4,j}} = j, \quad C^-_{F_2,R_{4,j}} = 1, \quad C^-_{F_2,R_5} = 1.$$

The semantic layer is the signed effect matrix $M_\Sigma = L = C^+ - C^-$, where the nonzero production entries are

$$C^+_{A,R_{1,i}} = 1, \quad C^+_{F_2,R_2} = 1, \quad C^+_{P,R_{3,i}} = C^+_{\delta,R_{3,i}} = 1,$$

$$C^+_{Q,R_{4,j}} = C^+_{\delta,R_{4,j}} = 1, \quad C^+_{Q,R_5} = C^+_{\delta,R_5} = 1.$$

The rule-interaction layer $M_r$ records five dependency families: parallel counting, phase-marker dependency, $A$-dependency, positive decision and negative decision. The control layer $M_\chi$ is the unsigned incidence matrix of the above priority path on the nine decision rules.

*Component ATDA descriptors.* For each incidence-type matrix $T$, we compute

$$\mathrm{ATDA}(T) = (n_T, m_T, r_T, \beta_0(T), \beta_1(T), \chi(T), \mathrm{Tor}\, H_0(T)).$$

The component descriptors, used here as static baselines for the active semantic analysis, are shown in Table 3.

**Table 3.** Component ATDA descriptors for the BCD experiment with $N = 8$.

| Layer | Matrix | $n$ | $m$ | $r$ | $\beta_0$ | $\beta_1$ | $\chi$ | Tor $H_0$ |
|---|---|---|---|---|---|---|---|---|
| Structural | $M_s$ | 6 | 6 | 6 | 0 | 0 | 0 | $\mathbb{Z}_2$ |
| Localization | $M_a = I_{18}$ | 18 | 18 | 18 | 0 | 0 | 0 | 0 |
| Enabling | $M_B = C^-$ | 14 | 18 | 11 | 3 | 7 | $-4$ | 0 |
| Rule interaction | $M_r$ | 18 | 5 | 5 | 13 | 0 | 13 | 0 |
| Control | $M_\chi$ | 9 | 8 | 8 | 1 | 0 | 1 | 0 |
| Semantics | $M_\Sigma = L$ | 14 | 18 | 12 | 2 | 6 | $-4$ | 0 |

The enabling matrix has $\beta_1(C^-) = 7$, reflecting rule-level degrees of freedom caused by different requirements sharing the same counting and phase objects. The semantic matrix has $\beta_0(L) = 2$, showing that semantic effects reduce the number of independent zero-dimensional classes. In this encoding, the only nonzero torsion entry appears in the structural MAREX support.

*Active semantic ATDA trajectories.* In this conference version, the active analysis is reported at the semantic level. The full active global matrix contains the remaining structural, localization, enabling, interaction and control layers, but its complete descriptor is left for an extended version.

The potential semantic matrix $L$ contains all rule effects, whereas the active semantic matrices restrict $L$ to the columns selected in each phase:

$$L^-_{\mathrm{count}} = L\big|_{\{R_{1,1},R_{1,3},R_{1,7},R_2\}}, \qquad L^-_{\mathrm{dec}} = L\big|_{\{R_{4,3}\}},$$

$$L^+_{\mathrm{count}} = L\big|_{\{R_{1,1},R_{1,3},R_{1,5},R_{1,7},R_2\}}, \qquad L^+_{\mathrm{dec}} = L\big|_{\{R_{3,4}\}}.$$

Their descriptors are given in Table 4.

*Interpretation.* The experiment illustrates the practical role of active ATDA. The potential matrix $L$ contains all BCD rule effects, but a computation uses only a configuration-dependent subset of them. During counting, the negative trajectory activates three neighbour-counting rules and $R_2$, whereas the positive trajectory activates four neighbour-counting rules and $R_2$. Hence $r(L^-_{\mathrm{count}}) = 4$, $r(L^+_{\mathrm{count}}) = 5$ and $\beta_0(L^-_{\mathrm{count}}) = 10$, $\beta_0(L^+_{\mathrm{count}}) = 9$.

**Table 4.** Active semantic ATDA descriptors for the two BCD trajectories.

| Case | Matrix | $n$ | $m$ | $r$ | $\beta_0$ | $\beta_1$ | $\chi$ | Tor $H_0$ |
|---|---|---|---|---|---|---|---|---|
| Potential | $L$ | 14 | 18 | 12 | 2 | 6 | $-4$ | 0 |
| Negative counting | $L^-_{\mathrm{count}}$ | 14 | 4 | 4 | 10 | 0 | 10 | 0 |
| Positive counting | $L^+_{\mathrm{count}}$ | 14 | 5 | 5 | 9 | 0 | 9 | 0 |
| Negative decision | $L^-_{\mathrm{dec}}$ | 14 | 1 | 1 | 13 | 0 | 13 | 0 |
| Positive decision | $L^+_{\mathrm{dec}}$ | 14 | 1 | 1 | 13 | 0 | 13 | 0 |

Thus ATDA detects the additional active semantic direction corresponding to the fourth active neighbour. In this small example, the rank difference coincides with the number of independent active semantic rule effects. The point is not that ATDA is needed to count active rules in this small case, but that the same descriptor remains meaningful for coupled or dependent rule effects, where counting active rules is no longer sufficient.

During the decision phase, priority collapses the admissible dynamics to one rule: $R_{4,3}$ in the negative case and $R_{3,4}$ in the positive case. Consequently,

$$r(L^-_{\mathrm{dec}}) = r(L^+_{\mathrm{dec}}) = 1,$$

which separates the parallel counting phase from the priority-controlled decision phase. The selected rule then determines the semantic output:

$$3A + F_2 \longrightarrow Q + \delta, \qquad 4A + F_2 \longrightarrow P + \delta.$$

Thus the active descriptors distinguish four operational aspects of the BCD computation: parallel counting, the phase transition through $F_2$, priority-controlled selection of $P$ or $Q$, and final dissolution encoded by $\delta$. The example shows that, even at the semantic level, ATDA is not only a static descriptor of the potential rule matrix; it records the incidence topology selected along a computation.

## 7 Conclusions and Future Work

We have introduced a multi-hypergraph incidence framework for membrane systems in which structural, localization, enabling, interaction, control and semantic layers are assembled into a global incidence architecture. ATWINs provide explicit homological models for the resulting incidence operators, while ATDA descriptors extract ranks, Betti numbers, Euler characteristics and torsion data (when the matrices are integer-valued) for comparison across component, active, aggregated and global levels.

The experiments are intentionally small and serve as proof-of-concept diagnostics of two distinct hierarchy levels. The channel-state example tests the component level: the structural communication support and channel-state control both produce interpretable one-dimensional homology classes. The BCD–MAREX example tests the active semantic level: the descriptors distinguish the negative trajectory

$$F_1 + E_1 + E_3 + E_7 \longrightarrow 3A + F_2 \longrightarrow Q + \delta$$

from the positive trajectory

$$F_1 + E_1 + E_3 + E_5 + E_7 \longrightarrow 4A + F_2 \longrightarrow P + \delta.$$

Together, these experiments illustrate the diagnostic capacity of the framework at two complementary levels. The staged strategy is deliberate: the first example tests whether component incidence matrices yield interpretable homology independently of runtime configuration, while the second tests whether active semantic restrictions capture trajectory-dependent changes not visible from the potential rule set alone. The complete analysis of the active global matrix, integrating all six layers, is left for future work.

The present construction is restricted to one-dimensional chain complexes. The resulting Betti numbers and torsion coefficients are therefore invariants of explicit incidence encodings, not intrinsic invariants of membrane systems independently of representation. Semantic aggregation is treated through descriptor comparison rather than homology transport, and the method is diagnostic rather than algorithmically optimized.

Future work will address full global matrix analysis, inter-layer coupling descriptors, incremental ATWIN computation, higher-dimensional incidence models for multiway synchronization and boundary processes, and homology transport under semantic aggregation. Larger multi-membrane benchmarks and integration with the Topological Scale Framework [13] will be used to study $M_{\mathrm{glob}}$, $M_{\mathrm{glob}}^{\mathrm{act}}(q_t)$ and $\widetilde{M}_{\mathrm{glob}}$ as part of time–scale topological dynamical systems.

ATWIN and ATDA should therefore be understood as a diagnostic topological layer complementing the standard operational semantics of membrane systems.

*Acknowledgements.* This work is presented as a scientific tribute to Professor Jesús Mario Pérez-Jiménez, whose foundational contributions and sustained leadership have been central to the development of membrane computing.

This publication is part of the Spanish R&D&I project PID2023-147795NA-I00, *Topological Scale-Space Analysis, Recognition and Classification of Hypergraphs: ATopEH*, funded by MICIU/AEI/10.13039/501100011033 and by ERDF/EU, "A way of making Europe".

This work is also part of the Spanish R&D&I project SAINEVRA (PID2023-151065OB-I00), funded by MCIN/AEI/10.13039/501100011033, by ERDF funds and by the European Union NextGenerationEU/PRTR.

## References

1. Păun, Gh.: Computing with membranes. J. Comput. Syst. Sci. 61(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
2. Păun, Gh.: Membrane Computing: An Introduction. Springer, Berlin (2002)
3. Păun, Gh., Pérez-Jiménez, M.J.: Recent computability models inspired from biology: DNA and membrane computing. THEORIA 18(1), 71–84 (2008)

4. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with active membranes on CUDA. Brief. Bioinform. 11(3), 313–322 (2010)
5. Dennunzio, A., Formenti, E., Manzoni, L., Margara, L., Menara, M.: Topological properties of deterministic P systems with active membranes. J. Membr. Comput. 5(4), 193–204 (2023)
6. Giavitto, J.-L., Michel, O.: The MGS language. Electron. Notes Theor. Comput. Sci. 59(4) (2002)
7. Giavitto, J.-L., Michel, O.: Topological structures in membrane computing. Fundam. Inform. (2004)
8. Huang, Y. et al: Soybean leaf disease identification based on a hypergraph cell membrane computing network model. Sci. Rep. 14, 6134 (2024)
9. Hatcher, A.: Algebraic Topology. Cambridge University Press (2002)
10. Munkres, J.R.: Topology. 2nd edn. Prentice Hall (2000)
11. Song, Y., Li, H., Chen, Q., Pérez-Jiménez, M.J.: The computational properties of P systems with mutative membrane structures. Inf. Comput. 303, 105277 (2025)
12. Freund, R., Păun, Gh., Pérez-Jiménez, M.J.: Tissue P systems with channel states. Theor. Comput. Sci. 330(1), 101–116 (2005)
13. Molina-Abril, H., Morón-Fernández, M.J., Benito-Marimón, M., Díaz-del-Río, F., Real, P.: Topological scale framework for hypergraphs. Appl. Math. Comput. 485, 128989 (2025)
14. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: Proceedings of the 34th ICML (2017)
15. Jiang, Y., Zhang, L., Wang, X.: Channel-state cell-like P systems and their computational properties. Fundam. Inform. (2022)
16. Wang, X., Zhang, Y., Peng, H.: Hypergraph-based spiking neural P systems. Expert Syst. Appl. (2023)
17. Cascado-Caballero et al.: A new approach for software-simulation of membrane systems using a multi-thread programming model. *Simul. Model. Pract. Theory* **136**, 103007 (2024).
18. Zhang, X., Liu, Y.: A new communication P system model based on hypergraph structures. Int. J. Found. Comput. Sci. 30(7), 1021–1040 (2019)
19. Liu, W., Huang, Y., Zhou, Q.: Hypergraph-based numerical neural-like P systems with repartition protocols. IEEE Trans. Parallel Distrib. Syst. 35(2), 317–330 (2024)
20. Martín-Vide, C., Păun, Gh., Pazos, J., Pérez-Jiménez, M.J.: Tissue P systems. Theor. Comput. Sci. 296(2), 295–326 (2003)
21. Cascado-Caballero, D., Díaz-del-Río, F., Cagigas-Muñiz, D., Rios-Navarro, A., Guisado-Lizar, J.-L., Pérez-Hurtado, I., Riscos-Núñez, A.: MAREX: A general purpose hardware architecture for membrane computing. *Inf. Sci.* **584**, 360–386 (2022).
22. Ibarra, O.H., Pérez-Jiménez, M.A., Păun, Gh.: Spiking neural P systems. Inf. Comput. 205(2), 234–252 (2007)
23. Păun, A., Păun, Gh.: The power of communication: P systems with symport/antiport. New Gener. Comput. 20, 295–305 (2002). `https://doi.org/10.1007/BF03037362`
24. Păun, Gh., Pazos, J., Pérez-Jiménez, M.J., Rodríguez-Patón, A.: Symport/antiport P systems with three objects are universal. Fundam. Inform. 64, 345–358 (2005)
25. Pérez-Jiménez, M.J., Sancho-Caparrini, F.: Tissue-like P systems with cell division. Int. J. Comput. Math. 83(8–9), 593–606 (2006)

26. González-Díaz, R., Real, P.: On the cohomology of 3D digital images. Discrete Appl. Math. 147(2–3), 245–263 (2005)
27. Pilarczyk, P., Real, P.: Computation of cubical homology, cohomology and cohomological operations via chain contractions. Adv. Comput. Math. 41, 253–275 (2015)
28. Edelsbrunner, H., Harer, J.: Persistent homology—a survey. In: Surveys on Discrete and Computational Geometry. Contemporary Mathematics, vol. 453, pp. 257–282. American Mathematical Society (2008)
29. Bretto, A.: Hypergraph Theory: An Introduction. Mathematical Engineering. Springer, Cham (2013)
30. Murata, T.: Petri nets: Properties, analysis and applications. Proc. IEEE 78(4), 541–580 (1989)

# Towards olive tree water stress classification with spiking neural P systems

David Orellana-Martín[1,3][0000−0002−2892−6775],
Agustín Riscos-Núñez[1,3][0000−0002−5409−3578], and
Alejandro Linares-Barranco[2,3][0000−0002−6056−740X]

[1] Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain
{dorellana,ariscosn}@us.es
[2] Robotic and Technology of Computers Lab, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain
alinares@atc.us.es
[3] SCORE Laboratory, I3US, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain

**Abstract.** In precision agriculture, water stress detection is an active research line because of its importance in efficient irrigation and improvement of crop management. In this work, the temporal data from fruit growth measurements under normal and water-stress conditions are encoded into events. The data are represented as multichannel spike sequences, making it ideal for artificial neural models where the information is encoded into spikes. Motivated by this representation, we introduce a framework based on reservoir computing with spiking neural P systems. The main contributions of this work are threefold: first, a direct interface between agricultural data and the input neurons of NSNP systems is provided; second, the temporal dynamics of an NSNP reservoir are used to construct trajectory representations suitable for classification through a trainable output layer; and third, a common evaluation protocol is established for the future comparison of the proposed framework with the ReckOn neuromorphic accelerator and other baseline models. The paper provides the methodological foundations for applying NSNP reservoir computing to event-based agricultural data and for studying its potential for olive tree water stress classification.

**Keywords:** Spiking neural P systems · Reservoir computing · Neurocomputing · Water stress classification.

## 1 Introduction

Availability of water is a critical factor for the development of crops, and an early detection of water stress can contribute to the improvement of the irrigation strategies and the efficient use of resources [3]. Methods based on the

response of the plant itself can provide information that is complementary to the environmental or soil variables.

Water stress affects the dynamics of fruit growth and the water relationship between fruits and leaves [2]. Therefore, the temporal evolution of olive size or growth can contain interesting information about the water stress of the tree. For this purpose, different sensors are usually placed on fruits and leaves to gather information about some of these variables [11].

All these measurements generate signals that are not only numerical, but depend on the evolution. From a computational point of view, these data are easy to transform into multichannel event sequences, which makes them suitable for analysis using spiking models of computation. This assessment is based on three main points: The temporal dimension can be preserved, only the relevant changes or events may be processed and the data are compatible with spiking neural networks [15].

This work was motivated by a previous work on ReckOn [4] in which a recurrent spiking neural network with on-chip learning is implemented. In this sense, an MPSoC/FPGA platform-based implementation is available, making it easier to train and infer by using the accelerator [8].

Spiking neural P systems [5] are models of computation within the framework of membrane computing inspired by the architecture and behavior of living cells, in the case of SNP systems, of biological neurons. In this model, spikes are discrete and evolve through rules that fire if specific conditions of the neurons are satisfied, and sent to neurons connected through synapses. Since their introduction, different variants of the basic SNP systems have been studied, including recent models focused on learning and adaptation [17]. A particular type of this model introduces nonlinear behavior in its rules, the so-called nonlinear SNP systems (in short, NSNP systems), where the rules operate on spikes with continuous values, and whose activation and firing methods can correspond to nonlinear functions [13].

Reservoir computing introduces a framework for reproducing the behavior of dynamic systems through recurrent structures that amplify the input and in which only the output weights are trainable [10]. In the framework of NSNP systems, they have been proposed as reservoir computing models for time series analysis, using the neurons of the model as reservoir units, and training only the output layer. In [18], the authors used this type of devices for time series classification.

While they have been used in that sense, neither the study of agricultural data with multichannel input sequences nor the methodological comparison with neuromorphic architectures has been addressed. In this work, we propose to use the event sequences directly as spike inputs, using NSNP systems as a reservoir. The proposal is to consider a dataset of this nature and define a clear interface between these data and SNP systems. After that, a framework to analyze these data by means of NSNP systems and a protocol to evaluate and compare the new model against other neuromorphic approaches such as ReckOn will be introduced.

The rest of the paper is organized as follows: Section 2 introduces some concepts about ReckOn, SNP systems and reservoir computing. In Section 3, the case study is described, followed by the proposed framework of NSNP systems. Next, an evaluation and comparison protocol is described, finishing with some conclusions and open research lines.

## 2 Background

In this section, some details about the three main technical components of the study are introduced.

### 2.1 Recurrent spiking neural networks and ReckOn

Spiking neural networks are part of the third generation of artificial neural networks, where information is sent by electric pulses to the connected neurons, and networks can be organized in a different way than classical neuron layers. Each neuron has an internal state related to the membrane potential that stores information from incoming pulses. When a certain condition is satisfied, usually a potential threshold, the neuron fires a spike that is transmitted to neighboring neurons. The information may be encoded not only in the number of spikes emitted by neurons, but also in the moment they are fired, the firing order among different neurons and the joint activity between different neurons. Usually, $s_i(t) \in \{0, 1\}$, where $s_i(t) = 1$ denotes that the neuron $i$ emits a spike at an instant $t$.

A recurrent spiking neural network allows neurons to provide feedback on their own output, providing a form of dynamic memory that can process temporal dependencies. Training these models may require storing states and propagating information back in time. Methods based on *eligibility traces*, such as *e-prop* [1] can approximate learning by means of local mechanisms that make them suitable for neuromorphic hardware.

The ReckOn architecture is a neuromorphic digital processor designed to emulate recurrent spiking neural networks and allows adapting the parameters of the network directly in the device [4]. In general, the architecture consists of an input layer, a recurrent spiking hidden layer and an output layer.

This accelerator can be integrated into programmable platforms that combine conventional computing and FPGA logic. In [8], ReckOn was integrated in an MPSoC platform that allows controlling the accelerator from a high-level environment and training on temporal data.

### 2.2 Spiking neural P systems

Spiking neural P systems are a non-deterministic and distributed model of computation within the framework of membrane computing introduced in [5]. A spiking neural P system $\Pi$ of degree $m$ is a tuple $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$ where:

1. $O$ is a singleton alphabet $\{a\}$ whose only element represents the spikes;
2. $\sigma_i = (n_i, R_i)$ is the description of the neuron $i$, where $n_i$ represents the number of spikes initially located in neuron $i$, and $R_i$ corresponds to the set of rules that act in the neuron $i$ that can be of the following forms:
   - $E/a^c \to a; d$, where $E$ is a regular expression, $c \geq 1, a^c \in L(E), d \in \mathbb{N}$ (firing rules);
   - $a^s \to \lambda$, where $s \geq 1$ (forgetting rules).
3. $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}, (i, i) \notin syn$ is the set of synapses between different neurons.
4. $in$ is the input neuron and $out$ is the output neuron.

A spiking neural P system (for short, SNP system) can be seen as a directed graph with $m$ nodes where $syn$ represents the edges of such a graph. Each node $i$ represents the neuron $\sigma_i$, that contains initially $n_i$ spikes.

A firing rule $r \equiv E/a^c \to a; d$ is applicable in neuron $\sigma_i$ at time $t$ if such a neuron has $k$ spikes, $k \geq c$, $a^k \in L(E)$, where $L(E)$ is the language associated to the regular expression $E$. When applied, $c$ spikes are removed from the neuron $\sigma_i$ and closes the neuron for $d$ steps; that is, if the rule is applied at instant $t$ and $d \geq 1$, then the neuron is closed in $t, t+1, \ldots, t+d-1$, then emits the spike and is open again at instant $t+d$. If $d = 0$, the neuron emits the spike immediately to the neurons $\sigma_j$ such that $(i, j) \in syn$. If a spike is transmitted to a closed neuron, such spike is lost. If $L(E) = \{a^c\}$, then $E$ can be omitted from such a rule. If $d = 0$, the delay can be omitted from such a rule.

A forgetting rule $r \equiv a^s \to \lambda$ is applicable in neuron $\sigma_i$ at time $t$ if such a neuron has $s$ spikes. When applied, $s$ spikes are removed from the neuron $\sigma_i$ and no new spikes are generated.

For each $R_i$, there cannot exist a forgetting rule $a^s \to \lambda$ such that there exists a firing rule $E/a^c \to a$ in such a neuron such that $a^s \in L(E)$. In each computation step, each open neuron having at least one applicable rule must apply exactly one rule. If more than one rule can be applied, then it is selected non-deterministically. Therefore, rules are applied in a sequential way within each neuron, while the neurons operate in parallel.

A configuration of a SNP system $C_t$ at an instant $t$ is a tuple $C_t = ((n_1(t), d_1(t)), \ldots, (n_m(t), d_m(t)))$, where $n_i(t)$ denotes the amount of spikes of neuron $\sigma_i$ at instant $t$ and $d_i(t)$ represents the remaining refractory time during which neuron $\sigma_i$ will be closed. The initial configuration is $C_0 = ((n_1, 0), \ldots, (n_m, 0))$. A *halting configuration* $C_t$ is a configuration in which no rule is applicable and $d_i(t) = 0$ for every $1 \leq i \leq m$.

A computation step $C_t \Rightarrow_{\Pi} C_{t+1}$ may change the contents of the system applying the rules as stated above. A computation $\mathcal{C} = (C_0, C_1, \ldots, C_n)$ of a SNP system $\Pi$ is a (possibly finite) sequence of configurations whose first element is the initial configuration $C_0$, and for each $1 \leq t \leq n$, $C_{t-1} \Rightarrow_{\Pi} C_t$. A *halting computation* is a computation where $C_n$ is a *halting configuration*, $n \in \mathbb{N}$.

The input sequence is, usually, a *spike train* that is represented through a sequence of zeroes and ones that is fed to the input neuron $in$ as follows: Let $u = u_1 u_2 \ldots u_p, u_t \in \{0, 1\}$ be the input sequence. Then a new spike is introduced

in neuron *in* at the $t$-th step if $u_t = 1$. The output neuron *out* evolves according to its rules. Whenever it fires, a spike is sent to the environment. Therefore, the output can be represented by a spike train $v = v_1 v_2 \dots v_q, v_t \in \{0, 1\}$, where $v_t = 1$ if and only if the neuron *out* sends a spike to the environment at step $t$. For the output, a common numerical output convention is to consider the difference of the moments where the first and the second spikes are sent to the environment; that is, if the first spike is sent at time $t_1$ and the second spike is sent at time $t_2$, then the answer is $t_2 - t_1$.

The first natural extension of this model is the extended spiking neural P systems, where firing and forgetting rules are unified and extended in such a way that more than one spike can be fired by means of the application of a rule, that is described as $E/a^c \to a^p; d, c \geq p \geq 0, c \neq 0$.

The main idea is that, instead of the classical behavior of weights in the synapses between neurons, the information propagates by means of the firing of certain rules. For a proper description of this model, we refer the reader to [5, 12].

If delays are not present, a configuration in this model can be described as $C_t = (x_1(t), \dots, x_m(t))$, where $x_i(t)$ represents the number of spikes of neuron $\sigma_i$ at an instant $t$. The application of the rules according to the semantics of the model provides a transition step, denoted by $C_t \Rightarrow_{\Pi} C_{t+1}$.

In NSNP systems, instead of using a discrete nature for spikes and a linear behavior on the evolution of spikes by the rules, some restrictions are relaxed and real continuous values associated with spikes can be used, as well as nonlinear functions can be associated with both the left-hand side and the right-hand side of the rules, described as $T \mid a^{g(u)} \to a^{f(u)}$, where $T$ is a threshold and $f(u)$ and $g(u)$,

$$f, g : \mathbb{R} \to \mathbb{R}_{\geq 0}, \tag{1}$$

are possibly nonlinear functions satisfying

$$g(u) \geq f(u) \geq 0, \text{for every } u \in \mathbb{R}, \tag{2}$$

that can be activated if the threshold is satisfied, where $g(u)$ is the value of consumed spikes from the neuron when the rule is applied, and $f(u)$ is the value of produced spikes when the rule is applied. Basically, the evolution of a neuron can be described as

$$x_i(t+1) = F_i(x_i(t), \{x_j(t) : (\sigma_j, \sigma_i) \in syn\}, u_i(t)), \tag{3}$$

providing a function $F_i$ that governs the input/output influence on the neuron $\sigma_i$. $F_i$ is not an additional component of the NSNP system, but a compact notation for the local transition function induced by the rules in $R_i$, the incoming synapses, and the external input. For instance, in a simplified case, $F_i$ may update the state of a neuron by subtracting the amount consumed by its selected rule and adding the spikes received from presynaptic neurons and the external input. Let each neuron have exactly one rule, and let

$$\delta_i(t) = \begin{cases} 1, \text{if neuron } \sigma_i \text{ fires at } t, \\ 0, \text{otherwise.} \end{cases} \tag{4}$$

Then, such function $F_i$ could be defined as

$$F_i(x_i(t), \{x_j(t)\}, u_i(t)) = x_i(t) - \delta_i(t) g_i(x_i(t)) + \sum_{j:(\sigma_j,\sigma_i)\in syn} \delta_j(t) f_j(x_j(t)) + u_i(t), \tag{5}$$

where $x_i(t)$ represents the state of $\sigma_i$, $\delta_i(t)g_i(x_i(t))$ the amount of consumed spikes, the summatory is the amount of received spikes and $u_i(t)$ is the external input.

This model is similar to spiking neural networks because of the information processing via spikes and temporal dynamics. However, instead of membrane potential, synaptic weights and update equations, NSNP systems are usually described with configurations, evolution rules and synapses. A configuration of an NSNP system is defined as $C_t = (x_1(t), \ldots, x_m(t))$, where $x_i(t)$ can take real values, the rules are applied the same way as in SNP systems, and the concepts of transition (or computational step) and computation are the same as in SNP systems. In this sense, we can think of the configuration at instant $t + 1$ as $C_{t+1} = (F_1(\ldots), \ldots, F_m(\ldots))$.

### 2.3 Reservoir computing with nonlinear spiking neural P systems

Reservoir computing is an area of computation where recurrent dynamical systems are used to transform input sequences into internal state sequences, using an input layer, a reservoir and an output layer. The main point of these models is that the only trainable layer is the output layer, making it easier to train than other fully trainable models [10]. The state of the reservoir usually evolves according to

$$\mathbf{x}(t+1) = F(\mathbf{x}(t), \mathbf{u}(t)), \tag{6}$$

where $\mathbf{u}(t)$ represents the input, $\mathbf{x}(t)$ represents the state of the reservoir and $F$ is the fixed recurrent dynamics. In the particular case in which the reservoir is an NSNP system, $F$ represents the global transition induced by the local functions $F_i$ of the reservoir neurons. The output is described as

$$\widehat{\mathbf{y}} = G(W_{out}, \mathbf{b}, \Phi(\mathbf{x}(1), \ldots, \mathbf{x}(T))), \tag{7}$$

where $\Phi$ transforms the reservoir trajectory $\mathbf{x}(1), \ldots, \mathbf{x}(T)$ into a fixed-dimensional sequence representation, $T$ denotes the number of time steps of the input sequence, $W_{out}$ is the trainable readout weight matrix, $\mathbf{b}$ is the trainable bias vector, and $G$ maps the resulting representation into the final prediction.

In [14], the authors introduce NSNP systems with a reservoir for time series classification, where the system receives an input and generates a succession of configurations. These configurations can be used to extract some features, and only the output layer is trained. Besides, in [18] the authors use NSNP systems as dynamic structures for time series classification. More recently, NSNP-based reservoir computing has also been used for time series forecasting [9].

## 3 Olive tree water stress case study

In this section, we describe some aspects of the dataset that will be analyzed as a case study. The observed variables are related to the temporal evolution of olive fruit growth [2]. The data were collected from 19 olive trees, divided into trees under normal conditions and trees under water stress. The data describe the growth of olive fruit through time [16, 7].

From this original dataset, the information is encoded as events (spikes) that will be used as input for the aforementioned models [15]. From the 79 samples of the dataset, 39 samples are in class 0 and 40 samples are in class 1. Taking into account that there are different signals for each time series, we need to encode the information into multichannel spike sequences in 20 channels, corresponding to the different event signals describing each sample. Once encoded, the sample can be described as

$$S^{(n)} = \left(e_1^{(n)}, \ldots, e_{K_n}^{(n)}\right), \tag{8}$$

where each event is

$$e_k^{(n)} = \left(c_k^{(n)}, t_k^{(n)}\right). \tag{9}$$

Here, $c_k^{(n)}$ identifies the channel, $t_k^{(n)}$ denotes the instant and $K_n$ is the number of events in sample $n$. Each sample has a label $y^{(n)} \in \{0, 1\}$, where 0 denotes normal conditions and 1 represents water stress conditions. The events in each sample are ordered according to their timestamps; that is,

$$t_1^{(n)} \leq t_2^{(n)} \leq \ldots \leq t_{K_n}^{(n)}. \tag{10}$$

Events occurring simultaneously in different channels may therefore share the same timestamp.

The event sequence can be transformed into a binary matrix

$$X^{(n)} = \left[x_i^{(n)}(t)\right] \in \{0, 1\}^{20 \times T}, \tag{11}$$

where

$$x_i^{(n)}(t) = \begin{cases} 1, \text{ if an event occurs in channel } i \text{ at instant } t, \\ 0, \text{ otherwise.} \end{cases} \tag{12}$$

These binary values represent the presence or absence of encoded events. Therefore, $x_i^{(n)}(t) = 0$ means that no event is generated in channel $i$ at instant $t$. Missing sensor observations must be treated during preprocessing.

## 4 A framework for processing agricultural event streams with SNP systems

In this section, a general description of an NSNP reservoir framework to analyze the dataset is provided. This framework is based on NSNP systems used as reservoirs introduced in [14].

At each time step, the input of the network is described as

$$\mathbf{u}^{(n)}(t) = \left(x_1^{(n)}(t), \ldots, x_{20}^{(n)}(t)\right)^\top. \tag{13}$$

Therefore, $\mathbf{u}^{(n)}(t) \in \{0,1\}^{20}$. Each component determines whether an event has occurred at the corresponding channel at time $t$. Given the input neurons $I = \{\sigma_1^{in}, \ldots, \sigma_{20}^{in}\}$, the mapping $\iota : \{1, \ldots, 20\} \to I$, where $\iota(i) = \sigma_i^{in}$, associates each channel with a specific input neuron. All samples start from the same initial configuration, i.e., $C_0^{(n)} = C_0$ for every $n$. The reservoir is reset to $C_0$ before processing each sample, since the samples are considered independent sequences. Therefore, the trajectory generated for a sample depends only on its input sequence and the fixed reservoir dynamics, and not on the state left by a previously processed sample.

At each instant $t$, the input vector $\mathbf{u}^{(n)}(t)$ is introduced into the system through the mapping $\iota$. In particular, if $x_i^{(n)}(t) = 1$, one spike is introduced into the corresponding input neuron $\sigma_i^{in}$, whereas no spike is introduced if $x_i^{(n)}(t) = 0$. We denote this input mechanism by $\mathcal{I}(\mathbf{u}^{(n)}(t))$.

Given the reservoir neurons $R = \{\sigma_1^R, \ldots, \sigma_N^R\}$, the functions and synapses are fixed throughout training and inference [14, 18]. Let

$$C_t^{(n)} = \left(z_1^{(n)}(t), \ldots, z_N^{(n)}(t)\right) \tag{14}$$

be the configuration of the reservoir for the sample $n$ at an instant $t$, where $z_j^{(n)}(t)$ represents the state of the neuron $j$ of the reservoir. The evolution of the configuration can be described as

$$C_t^{(n)} = \mathcal{F}_\Pi\left(C_{t-1}^{(n)}, \mathcal{I}\left(\mathbf{u}^{(n)}(t)\right)\right) \tag{15}$$

where $\mathcal{F}_\Pi$ represents the dynamics determined by the rules, functions and synapses of the NSNP system, $C_t^{(n)}$ is the new configuration and $\mathcal{I}\left(\mathbf{u}^{(n)}(t)\right)$ represents the events introduced in that instant. After the whole sample has been processed, the sequence

$$\mathcal{H}^{(n)} = \left(C_1^{(n)}, \ldots, C_T^{(n)}\right) \tag{16}$$

is generated. The initial configuration $C_0^{(n)}$ is not included in $\mathcal{H}^{(n)}$ since it represents the fixed reservoir state before processing the sample. Thus, $C_t^{(n)}$, for $1 \leq t \leq T$, represents the reservoir response after processing the input vector $\mathbf{u}^{(n)}(t)$. Since each configuration $C_t^{(n)}$ is a vector, $\mathcal{H}^{(n)}$ can be represented as

$$H^{(n)} = \begin{bmatrix} z_1^{(n)}(1) & \cdots & z_1^{(n)}(T) \\ \vdots & \ddots & \vdots \\ z_N^{(n)}(1) & \cdots & z_N^{(n)}(T) \end{bmatrix} \in \mathbb{R}^{N \times T}. \tag{17}$$

Let us recall that the objective of this reservoir is to transform a sequence of 20 channels into a higher dimensional nonlinear trajectory, potentially making the classes easier to separate.

Since each label corresponds to an entire sample, $H^{(n)}$ can be transformed into a fixed-dimensional vector

$$\phi^{(n)} = \Phi\left(H^{(n)}\right) \in \mathbb{R}^{d_\phi}, \tag{18}$$

where $\Phi$ is an operator that will transform the matrix. Let us describe some possibilities for this purpose.

- The representation is the final configuration; that is,

$$\phi^{(n)}_{final} = C^{(n)}_T. \tag{19}$$

  It is easy to implement, but it can lose information from the previous stages.
- The representation is the mean activity over the whole sequence; that is,

$$\phi^{(n)}_{mean} = \frac{1}{T}\sum_{t=1}^{T} C^{(n)}_t. \tag{20}$$

- The representation is the sum of the reservoir states; that is,

$$\phi^{(n)}_{sum} = \sum_{t=1}^{T} C^{(n)}_t. \tag{21}$$

- The sequence is divided into $L$ intervals $[1, T] = \mathcal{T}_1 \cup \ldots \cup \mathcal{T}_L$ and a representation is calculated by each interval; that is,

$$\phi^{(n)}_{window} = \left[\frac{1}{|\mathcal{T}_1|}\sum_{t\in\mathcal{T}_1} C^{(n)}_t, \ldots, \frac{1}{|\mathcal{T}_L|}\sum_{t\in\mathcal{T}_L} C^{(n)}_t\right]. \tag{22}$$

Although the last alternative may preserve temporal evolution better, the most appropriate representation should be determined experimentally.

Once the reservoir representation has been fixed, the transformed dataset is defined as

$$\mathcal{D}_\phi = \left\{\left(\phi^{(n)}, y^{(n)}\right)\right\}_{n=1}^{M}. \tag{23}$$

For training the output layer, each class label $y^{(n)} \in \{0, 1\}$ is represented by the target vector

$$\mathbf{y}^{(n)} = \begin{cases} (1,0)^\top & \text{if } y^{(n)} = 0, \\ (0,1)^\top & \text{if } y^{(n)} = 1. \end{cases} \tag{24}$$

The output layer receives $\phi^{(n)}$ and produces two different values, one for each class:

$$\mathbf{o}^{(n)} = W_{out}\phi^{(n)} + \mathbf{b}, \tag{25}$$

where $W_{out} \in \mathbb{R}^{2\times d_\phi}$ and $\mathbf{b} \in \mathbb{R}^2$ are the trainable weight matrix and bias vector of the output layer, respectively. Then

$$\mathbf{p}^{(n)} = \text{softmax}\left(\mathbf{o}^{(n)}\right). \tag{26}$$

The prediction is then

$$\widehat{y}^{(n)} = \arg\max_{c\in\{0,1\}} p_c^{(n)}. \tag{27}$$

In [14], ridge regression is used to learn the output layer and softmax for classification. Thus, this framework can follow the same approach:

$$(W_{out}^*, \mathbf{b}^*) = \underset{W,\mathbf{b}}{\text{argmin}} ||Y - W\Phi - \mathbf{b}1^\top||_F^2 + \lambda||W||_F^2, \tag{28}$$

where $\Phi \in \mathbb{R}^{d_\phi \times M_{\text{train}}}$ contains the representations of the training samples as columns,

$$Y = \left[\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(M_{\text{train}})}\right] \in \mathbb{R}^{2\times M_{\text{train}}} \tag{29}$$

contains their corresponding target vectors, $\mathbf{1} \in \mathbb{R}^{M_{\text{train}}}$ is the all-ones vector, and $\lambda$ controls the regularization.

The whole processing method of this framework is illustrated in Fig. 1.

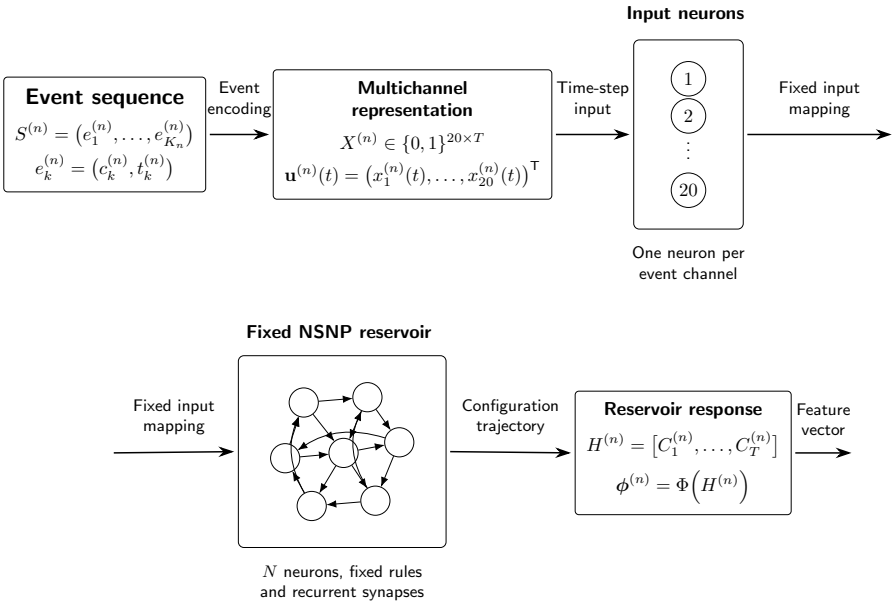
While the input $\mathbf{u}^{(n)}(1), \ldots, \mathbf{u}^{(n)}(T)$ is provided to both NSNP systems and ReckOn, the processing method is different.

## 5 A methodological comparison and evaluation protocol

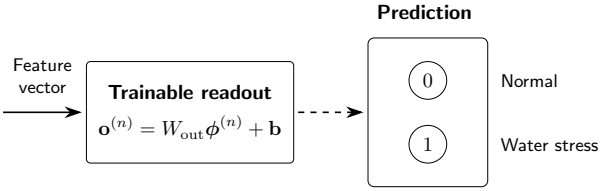
Given the fact that the input sequence is exactly the same, the idea is that the differences between the compared models arise not from the encoding but from different aspects of the models, such as the internal dynamics, the learning mechanism or the sequence representation. One of the main points here is that, while ReckOn adapts recurrent and output parameters during training, in the NSNP system approach only the output layer parameters are learned.

There are some questions that arise from the idea of this work. First, the main question is whether NSNP systems can be useful for water stress classification, that is the main objective of the framework. Within this framework, several design choices must be made in terms of parameters or functions. An interesting ingredient is the trajectory representation. In Section 4 different representations are introduced, but it remains to be determined whether any of them is capable of capturing the necessary information to obtain good results from this framework. From a comparative perspective, questions regarding metrics and parameters between the two models presented would be very useful for determining under which situation one of the frameworks should be selected over the other one.

A stratified group $K$-fold cross-validation process is proposed, where if several samples originate from the same olive tree, all of them must be assigned to the same fold. Each fold must contain both conditions (normal and water stress)




(a) Fixed processing pipeline.




(b) Trainable component.

Fig. 1: Overview of the proposed NSNP reservoir framework for olive tree water stress classification.

and the distribution among them must be reasonable. For the nested cross-validation, the outer loop should estimate the performance, while the inner loop should select the hyperparameters, only using the training data of each fold.

It can be formalized as

$$\Theta_m^{*(k)} = \underset{\Theta_m}{argmax} Score_{\,inner}\left(m, \Theta_m; \mathcal{D}_{train}^{(k)}\right), \tag{30}$$

and then

$$r_m^{(k)} = Score\left(m, \Theta_m^{*(k)}; \mathcal{D}_{test}^{(k)}\right), \tag{31}$$

where $m$ is the model, $\Theta_m$ are its hyperparameters, $k$ is the outer fold and $r_m^{(k)}$ is the performance on olive trees not used during the training.

In particular, the parameters of the NSNP system are the following:

$$\Theta_{NSNP} = (N, \rho, \Theta_{rules}, \Theta_{input}, \Phi, L, \lambda), \tag{32}$$

where

- $N$ is the number of neurons in the reservoir;
- $\rho$ is the density of connections;
- $\Theta_{rules}$ are the parameters of the functions and thresholds;
- $\Theta_{input}$ is the configuration of the input interface;
- $\Phi$ is the trajectory representation;
- $L$ is the number of temporal windows;
- $\lambda$ is the ridge regression regularization.

For ReckOn, the parameters would be:

- Parameters of the neuron dynamics;
- Thresholds;
- Input, recurrent and output learning rates;
- Epoch number;
- Other learning parameters.

In this sense, it is important that all the models use the same folds, and that the test sets are not used during the optimization.

The ReckOn and NSNP models should also be compared against a basic echo state network in order to see whether any advantage arises from the NSNP dynamics or simply because it is using a recurrent reservoir [6, 10]. Besides, some other models, such as logistic regression, random forest or support vector machines could be included as baselines representing different modelling perspectives.

Since the classes are balanced, *accuracy* can be used a metric, but also others such as *balanced accuracy*, *macro-F1* and *MCC* should be used. Besides, *sensitivity* to detect water stress and *specificity* to recognize normal conditions should also be reported.

Apart from that, other interesting metrics related to the computational capabilities should be studied, such as the training time, inference time, number of trainable parameters, number of neurons or memory usage, among others.

## 6 Conclusions and future work

This work has introduced a case study based on temporal information collected by different sensors and encoded into 20 channels. We have defined the basis for an NSNP framework for classifying samples as corresponding to normal conditions and water stress conditions. Besides, different ways to represent the trajectory have been defined. Moreover, an experimental protocol to evaluate the model and compare it with other models has been proposed. While this paper is conceptual and has not yet been experimentally validated, the framework

is motivated by previous evidence that NSNP systems can successfully process temporal information.

The immediate future work is to implement this framework in software to test a basic model with specific parameters, and develop a general experimental platform. Comparing all the proposed models would give us some ideas about the quality of the models studied. An automated comparison framework would be a very efficient way to compare other implemented models for this task. Apart from this general framework, another idea is to port the NSNP model implementation from software to hardware, making it possible to deploy these models on *edge computing* devices.

**Acknowledgments.** This work was funded by the Junta de Andalucía (Consejería de Universidad, Investigación e Innovación) and co-funded by the EU through the FEDER Andalucía 2021–2027 Programme under project DGP_PIDI_2024_00918. This work is also part of Project PID2025-174204NB-I00, funded by the Ministry of Science, Innovation and Universities (MICIU), the State Research Agency (AEI), DOI 10.13039/501100011033, and the European Regional Development Fund (ERDF), European Union. This work is part of a project that has received funding from the European Union's Horizon Europe Research and Innovation Programme under the Marie Skłodowska-Curie Actions (MSCA), Grant Agreement No. 101236749. This work was also supported by the Consejería de Universidad, Investigación e Innovación of the Junta de Andalucía through the PAIDI 2020 QUALIFICA programme, under grant QUAL21 008 USE (Smart Computer Systems Research and Engineering Lab).

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Bellec, G., Scherr, F., Hajek, E., Salaj, D., Legenstein, R., Maass, W.: A solution to the learning dilemma for recurrent networks of spiking neurons. Nature Communications **11**(1), 3625 (2020). https://doi.org/10.1038/s41467-020-17236-y
2. Fernandes, R.D.M., Cuevas, M.V., Díaz-Espejo, A., Hernández-Santana, V.: Effects of water stress on fruit growth and water relations between fruits and leaves in a hedgerow olive orchard. Agricultural Water Management **210**, 32–40 (2018). https://doi.org/10.1016/j.agwat.2018.07.028
3. Fernández, J.E.: Plant-based sensing to monitor water stress: Applicability to commercial orchards. Agricultural Water Management **142**, 99–109 (2014). https://doi.org/10.1016/j.agwat.2014.04.017
4. Frenkel, C., Indiveri, G.: ReckOn: A 28 nm sub-mm$^2$ task-agnostic spiking recurrent neural network processor enabling on-chip learning over second-long timescales. In: 2022 IEEE International Solid-State Circuits Conference (ISSCC). vol. 65, pp. 1–3. IEEE (2022). https://doi.org/10.1109/ISSCC42614.2022.9731734
5. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae **71**(2–3), 279–308 (2006). https://doi.org/10.3233/FUN-2006-712-308
6. Jaeger, H.: The "echo state" approach to analysing and training recurrent neural networks. Tech. Rep. GMD Report 148, German National Research Center for Information Technology (2001). https://doi.org/10.24406/publica-fhg-291111

7. Khosravi, A., Zucchini, M., Mancini, A., Neri, D.: Continuous third phase fruit monitoring in olive with regulated deficit irrigation to set a quantitative index of water stress. Horticulturae **8**(12), 1221 (2022). https://doi.org/10.3390/horticulturae8121221
8. Linares-Barranco, A., Prono, L., Legenstein, R., Indiveri, G., Frenkel, C.: Adaptive robotic arm control with a spiking recurrent neural network on a digital accelerator. In: 2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS). IEEE (2025). https://doi.org/10.1109/ICECS61496.2024.10849226
9. Long, L., Guo, C., Xiong, X., Peng, H., Wang, J.: A reservoir computing model based on nonlinear spiking neural p systems for time series forecasting. Applied Soft Computing **152**, 111644 (2024). https://doi.org/10.1016/j.asoc.2024.111644
10. Lukoševičius, M., Jaeger, H.: Reservoir computing approaches to recurrent neural network training. Computer Science Review **3**(3), 127–149 (2009). https://doi.org/10.1016/j.cosrev.2009.03.005
11. Marino, G., Scalisi, A., Guzmán-Delgado, P., Caruso, T., Marra, F.P., Lo Bianco, R.: Detecting mild water stress in olive with multiple plant-based continuous sensors. Plants **10**(1), 131 (2021). https://doi.org/10.3390/plants10010131
12. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford (2010). https://doi.org/10.1093/oso/9780199556670.001.0001
13. Peng, H., Lv, Z., Li, B., Luo, X., Wang, J., Song, X., Wang, T., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Nonlinear spiking neural P systems. International Journal of Neural Systems **30**(10), 2050008 (2020). https://doi.org/10.1142/S0129065720500082
14. Peng, H., Xiong, X., Wu, M., Wang, J., Yang, Q., Orellana-Martín, D., Pérez-Jiménez, M.J.: Reservoir computing models based on spiking neural P systems for time series classification. Neural Networks **169**, 274–281 (2024). https://doi.org/10.1016/j.neunet.2023.10.041
15. Roy, K., Jaiswal, A., Panda, P.: Towards spike-based machine intelligence with neuromorphic computing. Nature **575**(7784), 607–617 (2019). https://doi.org/10.1038/s41586-019-1677-2
16. Scalisi, A., Marino, G., Marra, F.P., Caruso, T., Lo Bianco, R.: A cultivar-sensitive approach for the continuous monitoring of olive (Olea europaea l.) tree water status by fruit and leaf sensing. Frontiers in Plant Science **11**, 340 (2020). https://doi.org/10.3389/fpls.2020.00340
17. Sosík, P., Paul, P., Ciencialová, L.: A survey on learning models of spiking neural membrane systems. Natural Computing **24**, 665–677 (2025). https://doi.org/10.1007/s11047-025-10026-9
18. Xiong, X., Wu, M., He, J., Peng, H., Wang, J., Long, X., Yang, Q.: Time series classification models based on nonlinear spiking neural P systems. Engineering Applications of Artificial Intelligence **129**, 107603 (2024). https://doi.org/10.1016/j.engappai.2023.107603

# Program-oriented models of computation and virus machines

David Orellana-Martín[1,2][0000−0002−2892−6775],
Luis Valencia-Cabrera[1,2][0000−0002−6576−9529], and
Mario J. Pérez-Jiménez[1,2][0000−0002−5055−0102]

[1] Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain
[2] SCORE Laboratory, I3US, Universidad de Sevilla, Avda. Reina Mercedes, s/n, 41012, Sevilla, Spain
{dorellana,lvalencia}@us.es

**Abstract.** Virus machines are a model of computation inspired by the transmission and replication mechanisms of viruses. When their computational completeness has been established through different characterizations, the relationship between their operational behavior and that of elementary imperative programming models has not been studied in detail. In this work, we establish a constructive correspondence between virus machines and GOTO programs. First, we provide a modular translation of the four instructions of the GOTO language into virus machines. Each variable is represented by a host, and the instructions are used to represent the instructions of the GOTO language. Conversely, we build a GOTO program that simulates the evolution of an arbitrary virus machine by representing the number of viruses in each host through a program variable. We prove that both transformations preserve the represented configurations and computational behavior up to finite sequences of internal simulation steps. Furthermore, upper bounds for the descriptional complexity of the resulting constructions are provided.

**Keywords:** Virus machines · GOTO programs · Simulation · Descriptional complexity.

## 1 Introduction

In 2015, virus machines were introduced as a model of computation inspired by the transmission and replication mechanisms of viruses [10]. It has been demonstrated through different methods that this model is universal in the sense of Turing machines: In [3] the universality was proved by simulating register machines. In [9] it was shown that virus machines can compute all partial recursive functions, and in [8] that they can generate all Diophantine sets, which coincide with the recursively enumerable sets by the MRDP theorem [5]. Besides, relationships with other models of computation can be found in [2] and [7]. In this sense,

operational, functional and generative points of view have been taken to prove the universality of this model of computation. However, there are no proofs at the moment concerning a direct operational correspondence to program-oriented models of computation (or elementary imperative programming languages).

In this sense, the GOTO language, introduced as the language $\mathcal{S}$ in [4] is ideal because it uses a minimal amount of basic instructions, and its structure is as simple as a sequence of instructions where variables and labels do not have complex semantics associated.

In this work, a modular translation from GOTO programs to virus machines and viceversa are provided, besies their corresponding descriptive complexity bounds. In general, a operational equivalence between both models is shown to be effective. The rest of the work is organized as follows: Section 2 provides an introduction to GOTO programs and virus machines, as well as some principles of operational simulation between models of computation and their configurations. Sections 3 and 4 provide the simulations from GOTO programs to virus machines and from virus machines to GOTO programs, respectively. Section 5 is devoted to discuss the equivalence between both models. In Section 6, a simple example is provided for the sake of demonstration. Finally, some conclusions and open research lines are described.

## 2 Preliminaries

In this section, the definition of the main models of computation studied in this work, GOTO programs and virus machines, are recalled. Besides, the operational simulation between different models of computation is described.

### 2.1 GOTO programs

A GOTO language is described as a sequence of instructions that follow specific semantics depending on the instructions executed. In [4], the programming language $\mathcal{S}$ is introduced, and the universality of the model is demonstrated. Now we recall the definition of a GOTO program.

**Definition 1.** *A GOTO program of size $s$ is a tuple*

$$\mathbf{P} = (I_1, \ldots, I_s),$$

*where each $I_k$ is an instruction of the following forms:*

- *Increment instruction:* $\mathtt{V} \leftarrow \mathtt{V} + \mathtt{1}$
- *Decrement instruction:* $\mathtt{V} \leftarrow \mathtt{V} - \mathtt{1}$
- *No operation instruction:* $\mathtt{V} \leftarrow \mathtt{V}$
- *Conditional instruction:* $\mathtt{IF\ V} \neq \mathtt{0\ GOTO\ L}$

*$V$ is an* arbitrary *variable of the program, whose domain is the set of natural numbers $\mathbb{N}$. Three different types of variables can be distinguished:*

- *Input variable:* $X_1, \ldots$
- *Auxiliary variable:* $Z_1, \ldots$
- *Output variable:* $Y$

*The value of variables* $Z_j$ *and* $Y$ *is* 0 *in the initial configuration.*

*L is a label of the program. Each instruction can be labelled as follows:* [L] `V` $\leftarrow$ `V` + `1`.

A GOTO program can be seen a top to bottom sequence of instructions such as the following:

```
    Y ← Y + 1
[A] IF X ≠ 0 GOTO B
    IF Y ≠ 0 GOTO E
[B] X ← X − 1
    Y ← Y + 1
    IF Y ≠ 0 GOTO A
```

This program calculates the function $f(x) = x + 1$; that is, the successor function.

The "state" of a GOTO program can be defined as the index of the next instruction to be executed and the sequence of values of the variables. Usually, the order for the variables is $Y, X_1, Z_1, X_2, Z_2, \ldots$ The configuration of a GOTO program $C_t = (k, v)$ at an instant $t$ is defined with $I_k$ as the current instruction to be executed and $v(V)$ is the current value of the variable $V$. If only one variable $X$ or $Z$ is used, the subscript is usually omitted. A configuration $C_t = (k, v)$ is a *halting* configuration if $k = s + 1$, where $s$ is the size of the program (i.e. the number of instructions)

If an instruction $I_k =$ `V` $\leftarrow$ `V` + `1` is executed, then the variable $V$ increases by one, and the following instruction is $I_{k+1}$. Then, $C_{t+1} = (k + 1, v')$ where $v'(V) = v(V) + 1$ and $v'(V') = v(V')$ if $V' \neq V$.

Let $\ominus$ be the following operator: $n \ominus 1 = n - 1$ if $n > 0$, and $n \ominus 1 = n$ if $n = 0$. If an instruction $I_k =$ `V` $\leftarrow$ `V` − `1` is executed, then the variable $V$ changes its value to $V \ominus 1$, and the following instruction is $I_{k+1}$. Then, $C_{t+1} = (k+1, v')$ where $v'(V) = v(V) \ominus 1$ and $v'(V') = v(V')$ if $V' \neq V$.

If an instruction $I_k =$ `V` $\leftarrow$ `V` is executed, then the following instruction is $I_{k+1}$. Then, $C_{t+1} = (k + 1, v)$.

If an instruction $I_k =$ `IF V` $\neq$ `0 GOTO L` is executed, then the following happens: If the variable $V$ is equal to 0, then $C_{t+1} = (k + 1, v)$; If the variable $V$ is not equal to 0, then $C_{t+1} = (c, v)$, where $c$ is the index of the first instruction labelled with $L$. If $L$ is not the label of any instruction, then $C_{t+1} = (s + 1, v)$, where $s$ is the size of the program. In the previous program, instruction $E$ is used for this purpose.

A transition of a GOTO program is made by the execution of one of the aforementioned instructions. A computation $\mathcal{C} = (C_0, C_1, \ldots)$ of a GOTO program $\mathbf{P}$ is a (possibly finite) sequence of configurations such that $C_0$ is the initial configuration of $\mathbf{P}$ and for each $t \in \mathbb{N}$, $C_t \Rightarrow_{\mathbf{P}} C_{t+1}$. A computation $\mathcal{C} = (C_0, C_1, \ldots, C_k)$ is called a *halting computation* if there exists a $k$ such that $C_k$ is a *halting configuration*; that is, $C_k = (s + 1, v)$.

The output of a GOTO program in a halting computation is determined by the value of the variable $Y$ at the last configuration; that is, at the halting configuration.

In the example, the instructions `IF Y` $\neq$ `0 GOTO L` work as dummy conditions, since from the first instruction it is known that the value of `Y` is greater than 0.

### 2.2 Virus machines

Virus machines were introduced in [10] as universal devices capable of simulating register machines if no restrictions are imposed, and weaker languages can be recognized if there are restrictions in the number of hosts, instructions or viruses present in the machines. Now we recall the definition of a virus machine.

**Definition 2.** *A virus machine of degree* $(p, q), p, q \geq 1$ *is a tuple*

$$\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out}),$$

*where:*

1. $\Gamma = \{v\}$ *is the singleton alphabet;*
2. $H = \{h_1, \ldots, h_p\}$ *and* $I = \{i_1, \ldots, i_q\}$ *are ordered sets such that* $v \in H \cup I$, $H \cap I = \emptyset$ *and* $h_{out} \in H$ *or* $h_{out} = h_0$*;*
3. $D_H = (H \cup \{h_{out}\}, E_H, w_H)$ *is a weighted directed graph, where* $E_H \subseteq H \times (H \cup \{h_{out}\})$, $(h, h) \notin E_H$ *for each* $h \in H$, *out-degree*$(h_{out}) = 0$ *and* $w_H$ *is a mapping from* $E_H$ *onto* $\mathbb{N} \setminus \{0\}$*;*
4. $D_I = (I, E_I, w_I)$ *is a weighted directed graph, where* $E_I \subseteq I \times I$, $w_I$ *is a mapping from* $E_I$ *onto* $\mathbb{N} \setminus \{0\}$ *and the out-degree of each node is less than or equal to* 2*;*
5. $G_C = (V_C, E_C)$ *is an undirected bipartite graph, where* $V_C = I \cup E_H$ *being* $\{I, E_H\}$ *the partition associated with it: every edge connects an element from* $I$ *with, at most, one arc from* $E_H$*;*
6. $n_j \in \mathbb{N} (1 \leq j \leq p)$.

A *virus machine* (VM, for short) $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$ of degree $(p, q)$, can be viewed as an ordered set of $p$ hosts labelled by elements from $H$, where each host $h_j$ initially contains $n_j$ *viruses*, and an ordered set of $q$ *control instruction units* labelled by elements from $I$. $h_{out}$ represents the output region, being a host if $h_{out} \in H$, and the *environment* of the machine if $h_{out} = h_0$. For the sake of simplicity, in this paper the environment will not be used. Arcs from the directed graph $D_H$ represent *transmission channels* through which viruses can transmit from one host $h_s$ (different from $h_{out}$) to another different host $h_{s'}$ or to the environment. The environment plays a passive role in virus machines, in the sense that it can only receive viruses from the device, but cannot take them back to it. Arcs from the directed graph $D_I$ represent *instruction transfer paths*. Finally, the undirected bipartite graph $G_C$ represents the *instruction-channel network* by which an edge $(i_j, (h_s, h_{s'}))$ indicates a control relationship between the instruction $i_j$ and the channel $(h_s, h_{s'})$.

Graphically, a virus machine of degree $(4, 6)$ can be depicted as a heterogeneous network consisting of three graphs, as illustrated in Fig. 1. Each host is represented as a rectangle and each instruction is represented with a circle. Arrows between hosts represent transmission channels, arrows between instructions represent instruction transfer paths and edges between instructions and channels represent instruction-channel connections. The weights of the edges are represented as positive numbers besides the edge (if the weight is 1, then it is not represented).

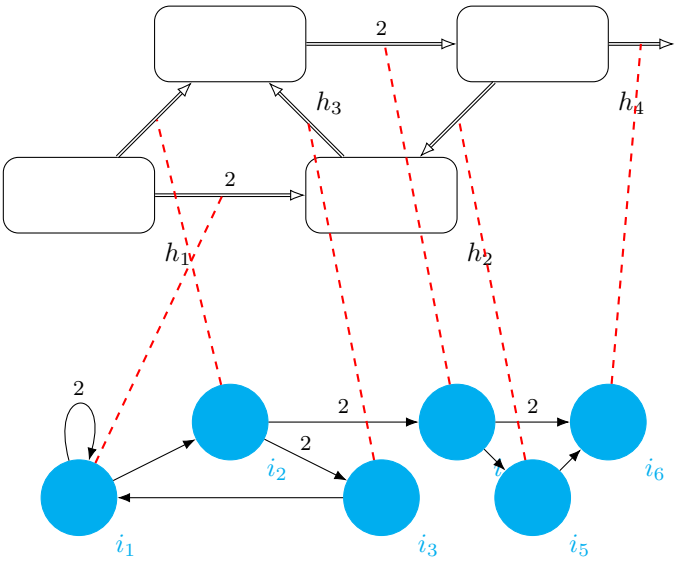



**Fig. 1.** Virus machine of degree $(4, 6)$

In the following paragraphs, the *semantics* of a virus machine will be described. First, we define the *instantaneous description* or a *configuration* $\mathcal{C}_t$ at an instant $t$ of a virus machine by a tuple $(a_1, \ldots, a_p, u, a_0)$, where $a_0, a_1, \ldots, a_p \in \mathbb{N}$ and $u \in I \cup \{\#\}$, where $\# \notin H \cup \{h_0\} \cup I$ is an object that characterizes a halting configuration. The meaning of $\mathcal{C}_t$ is the following: at an instant $t$, each host $h_j$ contains exactly $a_j$ viruses, the environment contains exactly $a_0$ viruses and, if $u \in I$, then the instruction $u$ will be activated at step $t + 1$, otherwise, then no instruction will be activated and the machine will halt. The initial configuration of a virus machine $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$ is $\mathcal{C}_0 = (n_1, \ldots, n_p, i_1, 0)$. A configuration $\mathcal{C}_t$ yields configuration $\mathcal{C}_{t+1}$ in one *transition step* or *computation step* if we can pass from $\mathcal{C}_t$ to $\mathcal{C}_{t+1}$ (and we denote it by $\mathcal{C}_t \Rightarrow_\Pi \mathcal{C}_{t+1}$) in the following form.

1. First, given that $\mathcal{C}_t$ is a non-halting configuration, we have that $u \in I$. Then, the control instruction unit $u$ is activated.

2. If $u$ is attached to a channel $(h_s, h_{s'})$, then the channel will be *opened* and:
   - If $a_s \geq 1$ then only one virus is consumed from host $h_s$ and $w_{s,s'}$ copies of $v$ are produced in the region $h_{s'}$.
   - If $a_s = 0$ then no viruses are consumed from host $h_s$ and no viruses are produced in the region $h_{s'}$
3. If $u$ is not attached to any channel then there is no transmission of viruses.
4. The next instruction to be executed is obtained as follows:
   - If *out-degree*$(u) = 2$ then there are two different instructions $u'$ and $u''$ such that $(u, u'), (u, u'') \in E_I$ (with weights $w_{u,u'}$ and $w_{u,u''}$, respectively).
     - If the instruction $u$ is attached to a channel $(h_s, h_{s'})$:
       - If $a_s \geq 1$ then the next instruction corresponds to the *highest* weight path $(max(\{w_{u,u'}, w_{u,u''}\}))$.
       - If $a_s = 0$ then the next instruction corresponds to the *lowest* weight path $(min(\{w_{u,u'}, w_{u,u''}\}))$.
       - In either case, if $w_{u,u'} = w_{u,u''}$, the next instruction is selected in a non-deterministic way.
     - If instruction $u$ is not attached to a channel, then the next instruction is selected in a non-deterministic way.
   - If *out-degree*$(u) = 1$ then the system behaves deterministically and $u'$ is the next instruction that verifies $(u, u') \in E_I$.
   - If *out-degree*$(u) = 0$ then $u = \#$ and $\mathcal{C}_{t+1}$ is a halting configuration.

A *computation* $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \ldots)$ of a virus machine $\Pi$ is a (possibly infinite) sequence of configurations such that $\mathcal{C}_0$ is the initial configuration of $\Pi$ and for each $t \in \mathbb{N}$, $\mathcal{C}_t \Rightarrow_\Pi \mathcal{C}_{t+1}$. A computation $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k)$ is called a *halting computation* if there exists a $k$ such that $\mathcal{C}_k$ is a *halting configuration*; that is, $u = \#$.

### 2.3 Operational simulation

In this work, a simulation module is a finite set of instructions and structures of the simulating model associated with the simulation of one instruction and (possibly) some structures of the simulated model. The boundaries of a simulation module are its entry and exit configurations; that is, the configurations corresponding to configurations of the simulated model. The intermediate configurations are considered internal simulation configurations and do not necessarily have a counterpart in the simulated model.

A model $M_1$ simulates a model $M_2$ if there exists an encoding of configurations such that each transition of $M_2$ is reproduced by a finite, non-empty sequence of transitions of $M_1$, while preserving the encoded data at the boundaries of the corresponding simulation modules. Two models of computation $M_1$ and $M_2$ are operationally equivalent if each model simulates the other; that is, if every transition of either model can be reproduced by a finite, non-empty sequence of transitions of the other model while preserving the correspondence between configurations at the boundaries of the simulation modules.

For this purpose, let us define the relation $\mathcal{R} \subseteq Conf(\mathbf{P}) \times Conf(\Pi)$, where $Conf(M)$ denotes the set of configurations of the model $M$, and $\mathcal{R}$ denotes that both computations represent the same "state". Then, we define $\approx = \mathcal{R} \cup \mathcal{R}^{-1}$; that is, $C_{M_1} \approx C_{M_2} \iff C_{M_2} \approx C_{M_1}$. Let $\Rightarrow_M \subseteq Conf(M) \times Conf(M)$ be a transition of the machine $M$, and $\Rightarrow_M^*$ be the natural extension where more than one transition can be taken.

## 3 Simulation of GOTO programs by means of virus machines

Let $P = (I_1, \ldots, I_m)$ be a GOTO program with variables $V_1, \ldots, V_n$. Then a virus machine $\Pi_P$ can be constructed by using different modules.

In this section, four modules simulating the four different instructions of the GOTO language will be introduced. First, it is interesting to point out that for each variable $V$ appearing in the program $\mathbf{P}$ a module like the one depicted in Fig. 2 will be constructed. Besides, each instruction in the program $\mathbf{P}$ that does not have any label, a label not appearing in any other instruction in the program will be assigned.

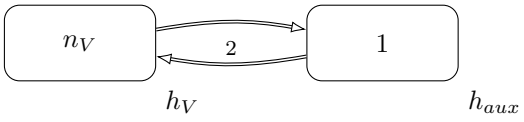



**Fig. 2.** Module for each variable $V$

Thus, we need a host $h_V$ for each variable $V$, an auxiliary host $h_{aux}$ whose initial number of viruses is 1. Besides, the value of the output variable $Y$ is represented by the output host $h_{out}$, and an special instruction $i_E$ represents the end of the GOTO program; that is, the instruction with index $s + 1$, where $s$ is the size of the program.

### 3.1 Increment instruction

Let the program $\mathbf{P}$ have the following instructions:

```
[K] ...
[L] V ← V + 1
[M] ...
```

Then, the module depicted in Fig. 3 will simulate the behavior of such instruction labeled by `L`. The idea is the following: when instruction $i_L$ is executed, it will move one virus from the host $h_{aux}$, replicating it into two viruses, and one of them will be back to $h_{aux}$ due to the application of instruction $i'_L$, letting $h_V$ have one more virus than before the execution of $i_L$, representing an increment of the variable $V$. Next instruction is $i_M$, corresponding with the next GOTO

instruction labeled by `M`. If `L` is the last instruction, then it will be connected to the instruction $i_E$.

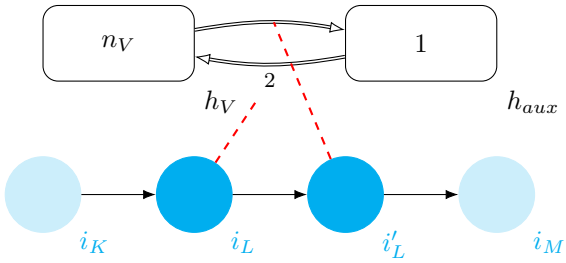



**Fig. 3.** Module for each instruction `[L] V ← V + 1`

**Lemma 1.** *Let $C = (L, v)$ be a configuration of $\mathbf{P}$, where the instruction labelled by $L$ is $\mathtt{V} \leftarrow \mathtt{V} + \mathtt{1}$. Starting the associated configuration of $\Pi_{\mathbf{P}}$, the module in Fig. 3 reaches, after two transitions, a configuration associated with the successor configuration of $C$.*

### 3.2 Decrement instruction

Let the program **P** have the following instructions:

```
[K] ...
[L] V ← V - 1
[M] ...
```

Then, the module depicted in Fig. 4 will simulate the behavior of such instruction labeled by `L`. The idea is the following: when instruction $i_L$ is executed, if there is a virus in host $h_V$, it will be moved to host $h_{aux}$, otherwise no viruses will be transported. This represents a decrement (if possible) of the variable $V$. Next instruction is $i_M$, corresponding with the next GOTO instruction labeled by `M`. If `L` is the last instruction, then it will be connected to the instruction $i_E$.

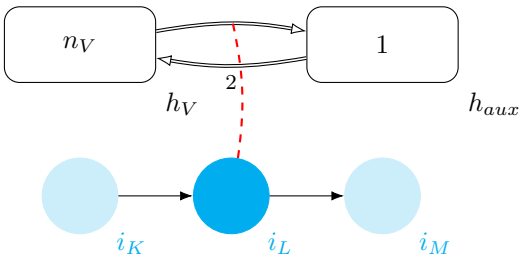



**Fig. 4.** Module for each instruction `[L] V ← V - 1`

**Lemma 2.** *Let $C = (L, v)$ be a configuration of* **P**, *where the instruction labelled by $L$ is* `V` $\leftarrow$ `V` $-$ `1`. *Starting the associated configuration of $\Pi_{\mathbf{P}}$, the module in Fig. 3 reaches, after one transition, a configuration associated with the successor configuration of $C$.*

### 3.3 No operation instruction

Let the program **P** have the following instructions:

```
[K] ...
[L] V ← V
[M] ...
```

Then, the module depicted in Fig. 5 will simulate the behavior of such instruction labeled by `L`. The idea is the following: when instruction $i_L$ is executed and it will let instruction $i_M$ to execute after that, corresponding with the next GOTO instruction labeled by `M`. If `L` is the last instruction, then it will be connected to the instruction $i_E$.

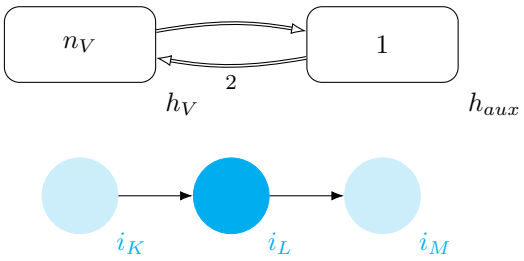



**Fig. 5.** Module for each instruction `[L] V` $\leftarrow$ `V`

**Lemma 3.** *Let $C = (L, v)$ be a configuration of* **P**, *where the instruction labelled by $L$ is* `V` $\leftarrow$ `V`. *Starting the associated configuration of $\Pi_{\mathbf{P}}$, the module in Fig. 3 reaches, after one transition, a configuration associated with the successor configuration of $C$.*

### 3.4 Conditional instruction

Let the program **P** have the following instructions:

```
[K] ...
[L] IF V ≠ 0 GOTO N
[M] ...
```

Then, the module depicted in Fig. 6 will simulate the behavior of such instruction labeled by `L`. The idea is the following: when instruction $i_L$ is executed, channel from $h_V$ to $h_{aux}$ is open. If there exists at least one virus, it means that $V \neq 0$, thus leading to the GOTO instruction labelled by `N`. In this case, the

simulation works in three steps, the first one being the instruction $i_L$ moving one virus from $h_V$ to $h_{aux}$. Since this virus needs to go back to $h_V$ (as there is neither increment nor decrement of $V$), first instruction $i'_L$ moves two viruses back to $h_V$, and instruction $i''_L$ moves one of these viruses back to $h_{aux}$, leading to the host $h_V$ to have the same amount of viruses as the beginning of the simulation of this GOTO instruction. Next instruction is $i_N$, corresponding with the next GOTO instruction labeled by `N`. In the case that host $h_V$ does not have any virus, it means that $V = 0$, thus the next GOTO instruction to be executed will be the instruction labelled by `M`. Since there was no movement of viruses, simply next instruction is $i_M$, corresponding with the next GOTO instruction labeled by `M`. If `L` is the last instruction, then it will be connected to the instruction $i_E$.

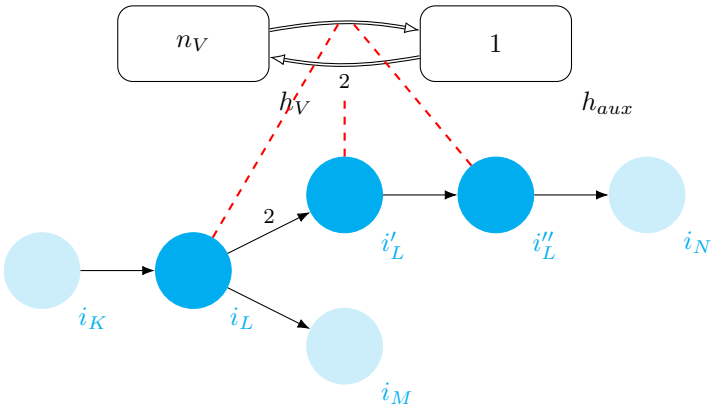



**Fig. 6.** Module for each instruction `[L] IF V ≠ 0 GOTO N`

**Lemma 4.** *Let $C = (L, v)$ be a configuration of $\mathbf{P}$, where the instruction labelled by $L$ is* `IF V ≠ 0 GOTO N`*. Starting the associated configuration of $\Pi_{\mathbf{P}}$, the module in Fig. 3 reaches, the configuration associated with the successor configuration of $C$ depending on the contents of $h_V$: If it has at least one virus, then it takes three transitions, and only one transition otherwise.*

**Theorem 1.** *For every GOTO program $\mathbf{P}$, a virus machine $\Pi_{\mathbf{P}}$ can be effectively constructed such that every computation of $\mathbf{P}$ is reproduced by $\Pi_{\mathbf{P}}$ at the boundaries of the simulation modules.*

### 3.5 Descriptional complexity of GOTO programs

Given a program $\mathbf{P}$, some aspects concerning the descriptional complexity are the following:

– The number of hosts would be $n+1$, where $n$ is the number of variables used in the program.

– The number of instructions is, at most, $3m + 1$, where $m$ is the number of instructions of $\mathbf{P}$, and an extra instruction $i_E$ representing the end of the program must be included.

## 4 Simulation of virus machines by means of GOTO programs

Let $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$. For each host $h \in H$, then the value of the variable $X_h$ is $n_h$.

The simulation of virus machines by means of GOTO programs will be carried out, as above, describing the modules of the program that will simulate all the possible constructions of instructions in virus machines. Particularly, three modules will be introduced for this purpose. Before the introduction of other modules, an initial instruction in the program is needed to include a variable that has value 1, that can be used for "non-conditional" `GOTO` instructions. $\mathtt{Z_{aux}} \leftarrow \mathtt{Z_{aux}}$ `+ 1`

### 4.1 Non-connected instruction

If there exists an instruction $i_k$ in the virus machine $\Pi$ such that it is not connected to any channel, as depicted in Fig. 7, then it can be simulated by the following piece of program:

$\mathtt{[i_k]}$ `IF` $\mathtt{Z_{aux}} \neq$ `0 GOTO` $\mathtt{i_1}$

The idea is the following: the GOTO instruction labelled by $i_k$ is executed and, since $Z_{aux} = 1$, the following instruction will be the instruction labelled by $\mathtt{i_1}$, without changing the value of any variable. If $i_k$ does not have a following instruction, then $i_l = E$, going to a halting configuration.

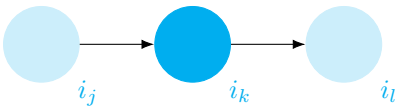



**Fig. 7.** Module for each non-connected instruction

### 4.2 Simple connected instruction

If there exists an instruction $i_k$ in the virus machine $\Pi$ such that it is associated with a channel from $h$ to $h'$ with weight $w$, as depicted in Fig. 8, then it can be simulated by the following piece of program:

$[\mathtt{i}_k]$ IF $\mathtt{X}_h \neq 0$ GOTO $\mathtt{i}'_k$
IF $\mathtt{Z}_{aux} \neq 0$ GOTO $\mathtt{i}_l$
$[\mathtt{i}'_k]$ $\mathtt{X}_h \leftarrow \mathtt{X}_h$ - 1
$\mathtt{X}_{h'} \leftarrow \mathtt{X}_{h'}$ + 1
...
$\mathtt{X}_{h'} \leftarrow \mathtt{X}_{h'}$ + 1
} $w$ times
IF $\mathtt{Z}_{aux} \neq 0$ GOTO $\mathtt{i}_l$

The idea is the following: the GOTO instruction labelled by $i_k$ is executed and the variable $X_h$ is checked. If $X_h = 0$, it means that the host $h$ does not have any virus, and then the instruction has been successfully simulated (no movement of viruses). Thus, the following GOTO instruction is executed, leading to the next GOTO instruction labeled by $\mathtt{i}_l$. Otherwise, it means that there is a transfer of viruses. In this sense, the GOTO instruction labelled by $\mathtt{i}'_k$ is executed, decreasing the variable $X_h$, corresponding to the number of viruses of the host $h$. After that, the instruction $\mathtt{X}_{h'} \leftarrow \mathtt{X}_{h'}$ + 1 is executed $w$ times, increasing that amount of times the value of variable $X_{h'}$, simulating the replication of viruses in host $h'$. The last instruction of this module corresponds to the selection of the next GOTO instruction labelled by $\mathtt{i}_l$. If $i_k$ does not have a following instruction, then $i_l = E$, going to a halting configuration.

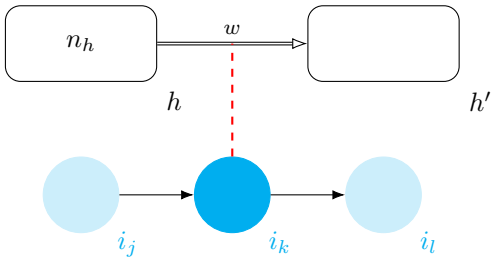



**Fig. 8.** Module for each simple connected instruction

### 4.3 Conditional connected instruction

If there exists an instruction $i_k$ in the virus machine $\Pi$ such that it is associated to a channel from $h$ to $h'$ with weight $w$, as depicted in Fig. 9, then it can be simulated by the following piece of program:

$[\mathtt{i}_k]$ IF $\mathtt{X}_h \neq 0$ GOTO $\mathtt{i}'_k$
IF $\mathtt{Z}_{aux} \neq 0$ GOTO $\mathtt{i}_m$
$[\mathtt{i}'_k]$ $\mathtt{X}_h \leftarrow \mathtt{X}_h$ - 1
$\mathtt{X}_{h'} \leftarrow \mathtt{X}_{h'}$ + 1
...
$\mathtt{X}_{h'} \leftarrow \mathtt{X}_{h'}$ + 1
} $w$ times
IF $\mathtt{Z}_{aux} \neq 0$ GOTO $\mathtt{i}_l$

The idea is the following: the GOTO instruction labelled by $i_k$ is executed and the variable $X_h$ is checked. If $X_h = 0$, it means that the host $h$ does not have any virus, and then the instruction has been successfully simulated. Thus, the following GOTO instruction is executed, leading to the next GOTO instruction labeled by $\mathtt{i}_1'$. This behavior corresponds to move to the instruction connected through the least-weight path. Otherwise, it means that there is a transfer of viruses. In this sense, the GOTO instruction labelled by $\mathtt{i}_k'$ is executed, decreasing the variable $X_h$, corresponding to the number of viruses of the host $h$. After that, the instruction $\mathtt{X_h} \leftarrow \mathtt{X_h} + \mathtt{1}$ is executed $w$ times, increasing that amount of times the value of variable $X_{h'}$, simulating the replication of viruses in host $h'$. The last instruction of this module corresponds to the selection of the next GOTO instruction labelled by $\mathtt{i}_1$, corresponding to the movement to the instruction connected through the highest-weight path.

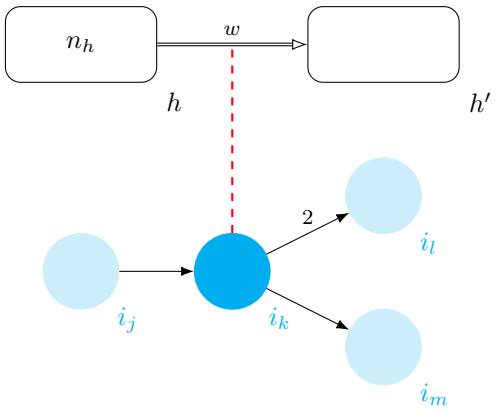



**Fig. 9.** Module for each conditional connected instruction

### 4.4 Descriptional complexity of virus machines

Given a virus machine $\Pi$, some aspects concerning the descriptional complexity are the following:

- The number of variables is $p + 1$, where $p$ is the number of hosts of $\Pi$.
- The number of labels is, at most, $2q$, where $q$ is the number of instructions of $\Pi$.
- The number of instructions is, at most, $1+(4+w_M)\cdot q$, where $q$ is the number of instructions of $\Pi$ and $w_M$ is the maximum weight of the hosts graph, and the first 1 is because of the first auxiliary instruction $\mathtt{Z_{aux}} \leftarrow \mathtt{Z_{aux}} + \mathtt{1}$.

**Theorem 2.** *For every virus machine $\Pi$, a GOTO program $\mathbf{P}_\Pi$ can be effectively constructed such that every transition of $\Pi$ is simulated by a finite sequence of instructions of $\mathbf{P}_\Pi$.*

## 5 Operational equivalence

In this section, we discuss the operational equivalence between virus machines and GOTO programs.

**Theorem 3.** *GOTO programs and virus machines are operationally equivalent.*

*Proof.* First, we demonstrate that for each GOTO program $\mathbf{P}$ there exists a virus machine $\Pi_{\mathbf{P}}$ that simulates its behavior. With the previously defined modules, we can build the virus machine $\Pi_{\mathbf{P}}$ that simulates the behavior from $\mathbf{P}$. Let $C = (L, v)$ be a configuration from $\mathbf{P}$. Then, there exists a configuration $C_{\Pi_{\mathbf{P}}} \approx C_{\mathbf{P}}$ such that $C_{\Pi_{\mathbf{P}}} \Rightarrow^*_{\Pi_{\mathbf{P}}} C'_{\Pi_{\mathbf{P}}}$ and $C'_{\Pi_{\mathbf{P}}} \approx C'_{\mathbf{P}}$, where $C_{\mathbf{P}} \Rightarrow_{\mathbf{P}} C'_{\mathbf{P}}$.

The opposite direction is similarly proved. With the previously defined modules, we can build the GOTO program $\mathbf{P}_\Pi$ that simulates the behavior from $\Pi$. Let $C = (n_1, \ldots, n_k, i, n_0)$ be a configuration from $\Pi$. Then, there exists a configuration $C_{\mathbf{P}_\Pi} \approx C_\Pi$ such that $C_{\mathbf{P}_\Pi} \Rightarrow^*_{\mathbf{P}_\Pi} C'_{\mathbf{P}_\Pi}$ and $C'_{\mathbf{P}_\Pi} \approx C'_\Pi$, where $C_\Pi \Rightarrow_\Pi C'_\Pi$.

**Corollary 1.** *Let $\mathbf{P}$ be a GOTO program and $\Pi_{\mathbf{P}}$ the virus machine simulating the program using the construction presented in Section 3. Let $C_0^{\mathbf{P}}$ and $C_0^{\Pi_{\mathbf{P}}}$ be the corresponding initial configurations; that is:*

$$C_0^{\mathbf{P}} \approx C_0^{\Pi_{\mathbf{P}}}.$$

*If*

$$C_0^{\mathbf{P}} \Rightarrow^*_{\mathbf{P}} C_h^{\mathbf{P}},$$

*where $C_h^{\mathbf{P}}$ is a halting configuration of $\mathbf{P}$, then there exists a halting computation*

$$C_0^{\Pi_{\mathbf{P}}} \Rightarrow^*_{\Pi_{\mathbf{P}}} C_h^{\Pi_{\mathbf{P}}},$$

*where $C_h^{\Pi_{\mathbf{P}}}$ is a halting configuration of $\Pi_{\mathbf{P}}$ such that*

$$C_h^{\mathbf{P}} \approx C_h^{\Pi_{\mathbf{P}}}.$$

*Analogously, let $\Pi$ be a virus machine and $\mathbf{P}_\Pi$ the GOTO program simulating the virus machine using the construction presented in Section 4. If $C_0^\Pi \approx C_0^{\mathbf{P}_\Pi}$ and*

$$C_0^\Pi \Rightarrow^*_\Pi C_h^\Pi$$

*is a halting computation of $\Pi$, then there exists a computation*

$$C_0^{\mathbf{P}_\Pi} \Rightarrow^*_{\mathbf{P}_\Pi} C_h^{\mathbf{P}_\Pi},$$

*for some halting configuration $C_h^{\mathbf{P}_\Pi}$ satisfying*

$$C_h^\Pi \approx C_h^{\mathbf{P}_\Pi}.$$

Since both simulations have internal configurations that do not correspond to any configuration from the other model, we have constructed a weak bisimulation between both models.

## 6 An example

For the sake of demonstration, the GOTO program **P** presented in Section 2 will be transformed into a virus machine simulating it, and this virus machine will be converted back into a GOTO program. First, for each instruction that does not have a label, we need to include a non-appearing label.

[C$_1$] Y $\leftarrow$ Y + 1
[A] IF X $\neq$ 0 GOTO B
[C$_2$] IF Y $\neq$ 0 GOTO E
[B] X $\leftarrow$ X − 1
[C$_3$] Y $\leftarrow$ Y + 1
[C$_4$] IF Y $\neq$ 0 GOTO A

This program is simulated by the virus machine $\Pi_{\mathbf{P}}$ represented in Figure 10, where the initial instruction is $i_{C_1}$ and $X$ is the initial value of the variable X. The number of hosts is $3 = 2 + 1$, where 2 is the number of variables used throughout the program **P**, and the number of instructions is $15 \leq 3 \cdot 6 + 1 = 19$, where 6 is the size of the program. The instructions of the virus machine corresponding to each original instruction from the GOTO program can be found within distinguished dashed rectangles in the figure. Take into account that the maximum number of instructions would be achieved if every instruction of **P** is a conditional instruction.

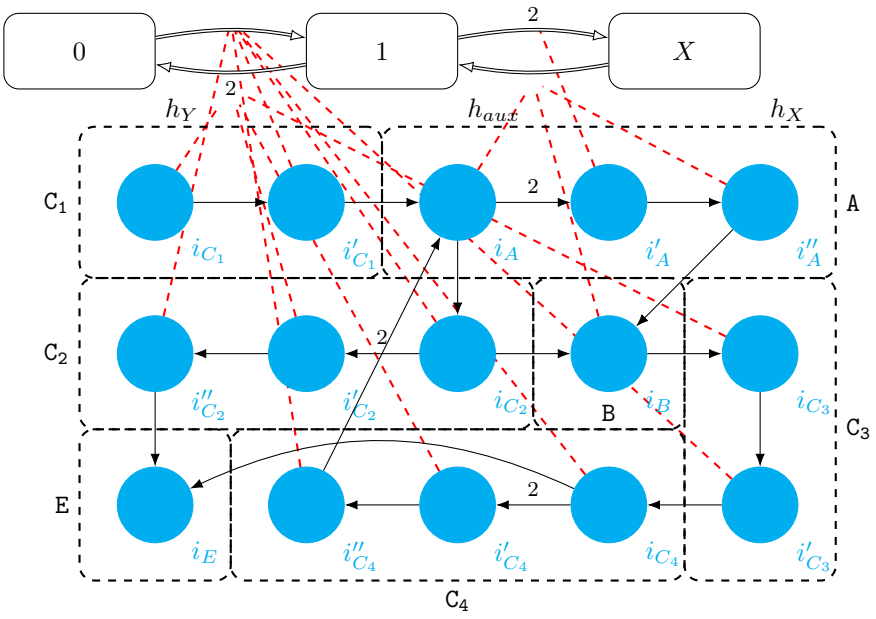
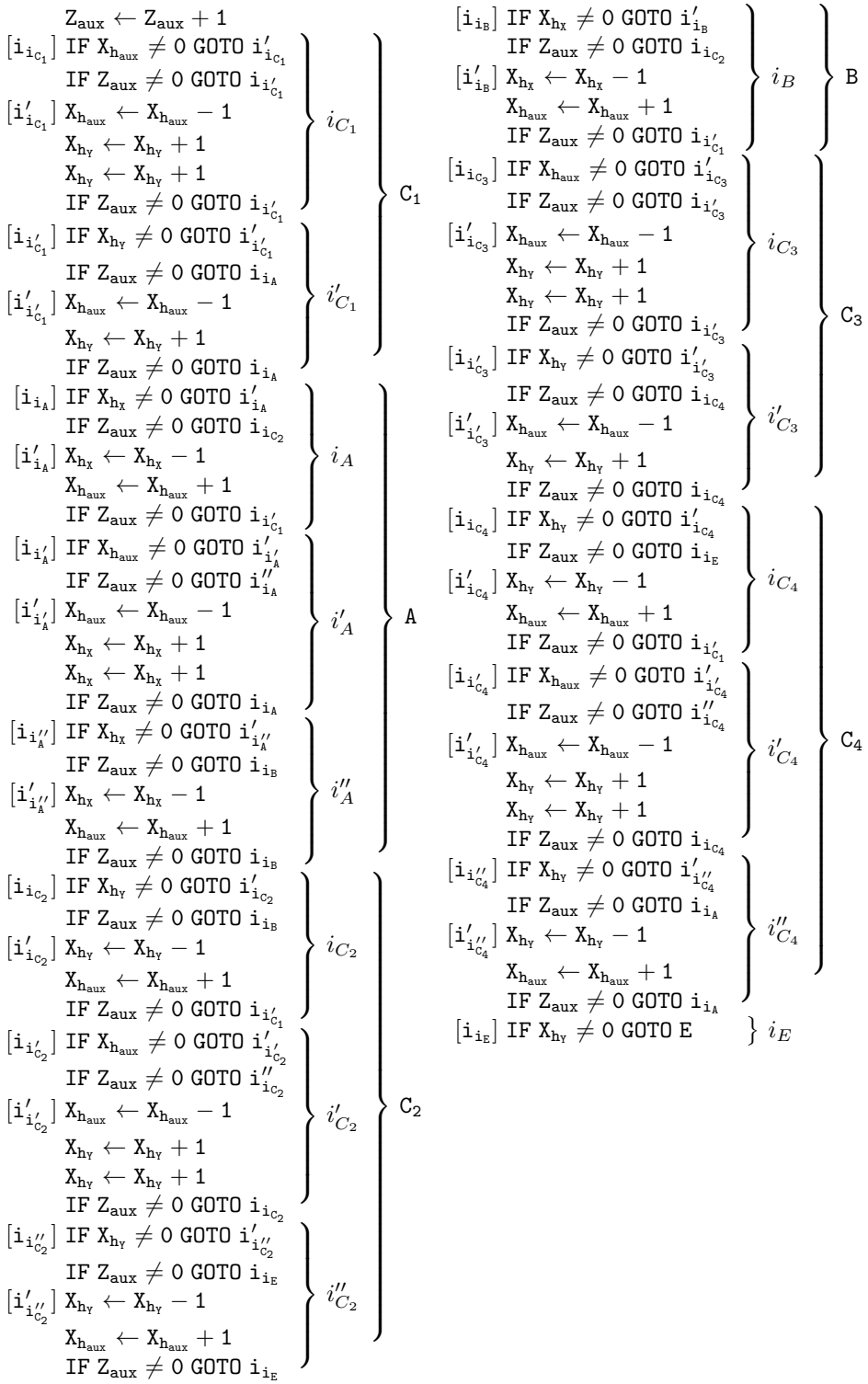



**Fig. 10.** Virus machine simulating program **P**

The virus machine represented in Figure 10 can be simulated by a GOTO program $\mathbf{P}_{\Pi_{\mathbf{P}}}$, that in fact would be simulating the original GOTO program.


Z_aux ← Z_aux + 1
[i_{i_{C_1}}] IF X_{h_aux} ≠ 0 GOTO i'_{i_{C_1}}
IF Z_aux ≠ 0 GOTO i_{i'_{C_1}}
[i'_{i_{C_1}}] X_{h_aux} ← X_{h_aux} − 1
X_{h_Y} ← X_{h_Y} + 1
X_{h_Y} ← X_{h_Y} + 1
IF Z_aux ≠ 0 GOTO i_{i'_{C_1}}
i_{C_1}
[i_{i'_{C_1}}] IF X_{h_Y} ≠ 0 GOTO i'_{i'_{C_1}}
IF Z_aux ≠ 0 GOTO i_{i_A}
[i'_{i'_{C_1}}] X_{h_aux} ← X_{h_aux} − 1
X_{h_Y} ← X_{h_Y} + 1
IF Z_aux ≠ 0 GOTO i_{i_A}
i'_{C_1}
C_1
[i_{i_A}] IF X_{h_X} ≠ 0 GOTO i'_{i_A}
IF Z_aux ≠ 0 GOTO i_{i_{C_2}}
[i'_{i_A}] X_{h_X} ← X_{h_X} − 1
X_{h_aux} ← X_{h_aux} + 1
IF Z_aux ≠ 0 GOTO i_{i'_{C_1}}
i_A
[i_{i'_A}] IF X_{h_aux} ≠ 0 GOTO i'_{i'_A}
IF Z_aux ≠ 0 GOTO i''_{i_A}
[i'_{i'_A}] X_{h_aux} ← X_{h_aux} − 1
X_{h_X} ← X_{h_X} + 1
X_{h_X} ← X_{h_X} + 1
IF Z_aux ≠ 0 GOTO i_{i_A}
i'_A
[i_{i''_A}] IF X_{h_X} ≠ 0 GOTO i'_{i''_A}
IF Z_aux ≠ 0 GOTO i_{i_B}
[i'_{i''_A}] X_{h_X} ← X_{h_X} − 1
X_{h_aux} ← X_{h_aux} + 1
IF Z_aux ≠ 0 GOTO i_{i_B}
i''_A
A
[i_{i_{C_2}}] IF X_{h_Y} ≠ 0 GOTO i'_{i_{C_2}}
IF Z_aux ≠ 0 GOTO i_{i_B}
[i'_{i_{C_2}}] X_{h_Y} ← X_{h_Y} − 1
X_{h_aux} ← X_{h_aux} + 1
IF Z_aux ≠ 0 GOTO i_{i'_{C_1}}
i_{C_2}
[i_{i'_{C_2}}] IF X_{h_aux} ≠ 0 GOTO i'_{i'_{C_2}}
IF Z_aux ≠ 0 GOTO i''_{i_{C_2}}
[i'_{i'_{C_2}}] X_{h_aux} ← X_{h_aux} − 1
X_{h_Y} ← X_{h_Y} + 1
X_{h_Y} ← X_{h_Y} + 1
IF Z_aux ≠ 0 GOTO i_{i_{C_2}}
i'_{C_2}
[i_{i''_{C_2}}] IF X_{h_Y} ≠ 0 GOTO i'_{i''_{C_2}}
IF Z_aux ≠ 0 GOTO i_{i_E}
[i'_{i''_{C_2}}] X_{h_Y} ← X_{h_Y} − 1
X_{h_aux} ← X_{h_aux} + 1
IF Z_aux ≠ 0 GOTO i_{i_E}
i''_{C_2}
C_2
[i_{i_B}] IF X_{h_X} ≠ 0 GOTO i'_{i_B}
IF Z_aux ≠ 0 GOTO i_{i_{C_2}}
[i'_{i_B}] X_{h_X} ← X_{h_X} − 1
X_{h_aux} ← X_{h_aux} + 1
IF Z_aux ≠ 0 GOTO i_{i'_{C_1}}
i_B
B
[i_{i_{C_3}}] IF X_{h_aux} ≠ 0 GOTO i'_{i_{C_3}}
IF Z_aux ≠ 0 GOTO i_{i'_{C_3}}
[i'_{i_{C_3}}] X_{h_aux} ← X_{h_aux} − 1
X_{h_Y} ← X_{h_Y} + 1
X_{h_Y} ← X_{h_Y} + 1
IF Z_aux ≠ 0 GOTO i_{i'_{C_3}}
i_{C_3}
[i_{i'_{C_3}}] IF X_{h_Y} ≠ 0 GOTO i'_{i'_{C_3}}
IF Z_aux ≠ 0 GOTO i_{i_{C_4}}
[i'_{i'_{C_3}}] X_{h_aux} ← X_{h_aux} − 1
X_{h_Y} ← X_{h_Y} + 1
IF Z_aux ≠ 0 GOTO i_{i_{C_4}}
i'_{C_3}
C_3
[i_{i_{C_4}}] IF X_{h_Y} ≠ 0 GOTO i'_{i_{C_4}}
IF Z_aux ≠ 0 GOTO i_{i_E}
[i'_{i_{C_4}}] X_{h_Y} ← X_{h_Y} − 1
X_{h_aux} ← X_{h_aux} + 1
IF Z_aux ≠ 0 GOTO i_{i'_{C_1}}
i_{C_4}
[i_{i'_{C_4}}] IF X_{h_aux} ≠ 0 GOTO i'_{i'_{C_4}}
IF Z_aux ≠ 0 GOTO i''_{i_{C_4}}
[i'_{i'_{C_4}}] X_{h_aux} ← X_{h_aux} − 1
X_{h_Y} ← X_{h_Y} + 1
X_{h_Y} ← X_{h_Y} + 1
IF Z_aux ≠ 0 GOTO i_{i_{C_4}}
i'_{C_4}
[i_{i''_{C_4}}] IF X_{h_Y} ≠ 0 GOTO i'_{i''_{C_4}}
IF Z_aux ≠ 0 GOTO i_{i_A}
[i'_{i''_{C_4}}] X_{h_Y} ← X_{h_Y} − 1
X_{h_aux} ← X_{h_aux} + 1
IF Z_aux ≠ 0 GOTO i_{i_A}
i''_{C_4}
C_4
[i_{i_E}] IF X_{h_Y} ≠ 0 GOTO E
i_E

The virus machine $\Pi_{\mathbf{P}}$ is simulated by the program $\mathbf{P}_{\Pi_{\mathbf{P}}}$, where the value of $\mathtt{X_i}$ in the initial configuration is the number of viruses in the initial configuration of the host $i$ from $\Pi_{\mathbf{P}}$; that is, for $\mathtt{X}_{\mathtt{h_Y}} = Y = 0, \mathtt{X}_{\mathtt{h_X}} = X$ and $\mathtt{X}_{\mathtt{h}_{\mathtt{aux}}} = 1$. The number of variables is $4 = 3 + 1$, where 3 is the number of hosts of $\Pi_{\mathbf{P}}$, the number of labels is $29 \leq 2 \cdot 15 = 30$, where 15 is the number of instructions of the virus machine, and the number of instructions is $77 \leq 1 + (4+2) \cdot 15 = 91$, where 15 is the number of instructions of the virus machines and 2 is the maximum weight of the hosts graph.

As it can be seen, the number of resources for the new program is greater than in the original one. The number of variables has been increased from 2 to 4, the number of labels has been increased from 6 to 29 and the number of instructions has been increased from 6 to 77. But since $\mathbf{P}_{\Pi_{\mathbf{P}}}$ can simulate $\Pi_{\mathbf{P}}$, and $\Pi_{\mathbf{P}}$ can simulate $\mathbf{P}$, then $\mathbf{P}_{\Pi_{\mathbf{P}}}$ simulates $\mathbf{P}$ in the following sense: A block `K` (indicated in the right brackets) simulates a single instruction from the original program. When the program finishes, the variable $\mathtt{X}_{\mathtt{h_X}}$ indicates the value of the original variable `X`, and $\mathtt{X}_{\mathtt{h_Y}}$ indicates the value of the original variable `Y`.

## 7 Conclusions and future work

In this work, we have obtained a constructive and bidirectional correspondence between virus machines and GOTO programs, preserving the number of variables and hosts in a linear way. One of the key points of the work is the modularity of the construction, leaving room for new modules from extensions of the studied models. The results show that the time increment for each simulation is always linear in terms of the size of the original model. In general, a program oriented point of view of virus machines has been provided in this work.

As future work, some extensions of each model could be studied to improve the performance of the simulations. Besides, normal forms of these simulations could be stablished [1]. It would also be interesting to develop an automatic translator from virus machines to GOTO programs and back, taking advantage of the simulator of virus machines [6].

**Acknowledgments.** This work is part of Project PID2025-174204NB-I00, funded by the Ministry of Science, Innovation and Universities (MICIU), the State Research Agency (AEI), DOI 10.13039/501100011033, and the European Regional Development Fund (ERDF), European Union. We want to acknowledge the present work to Mario J. Pérez-Jiménez. We still remember our insightful discussions about this topic from last year. Thank you very much for showing us the wonderful part of science: Sharing it with others with passion, and living it as a family. Your legacy will always be alive.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Ramírez-de Arellano, A., Cabarle, F.G.C., Orellana-Martín, D., Pérez-Jiménez, M.J.: On the normal forms and computational power of

Virus Machines. Theoretical Computer Science **1083**, 116139 (2026). https://doi.org/10.1016/j.tcs.2026.116139
2. Ramírez-de Arellano, A., Orellana-Martín, D., Pérez-Jiménez, M.J.: Bridges between spiking neural membrane systems and Virus Machines. International Journal of Neural Systems **34**(6), 2450034 (2024). https://doi.org/10.1142/S0129065724500345
3. Chen, X., Pérez-Jiménez, M.J., Valencia-Cabrera, L., Wang, B., Zeng, X.: Computing with viruses. Theoretical Computer Science **623**, 146–159 (2016). https://doi.org/10.1016/j.tcs.2015.12.006
4. Davis, M.D., Sigal, R., Weyuker, E.J.: Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science. Morgan Kaufmann, 2 edn. (1994). https://doi.org/10.1016/C2009-0-21176-7
5. Matiyasevich, Y.V.: Enumerable sets are diophantine. Soviet Mathematics Doklady **11**(2), 354–357 (1970)
6. Orellana-Martín, D., Ramírez-de Arellano, A., Pérez-Jiménez, M.J.: Simulating and validating Virus Machines. Natural Computing **24**(2), 311–320 (2025). https://doi.org/10.1007/s11047-025-10013-0
7. Orellana-Martín, D., Romero-Jiménez, Á., Riscos-Núñez, A., Pérez-Jiménez, M.J.: From Petri Nets to Virus Machines. In: Formenti, E., Durand-Lose, J. (eds.) Machines, Computations, and Universality, Lecture Notes in Computer Science, vol. 15270, pp. 103–114. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-81202-6_7
8. Romero-Jiménez, Á., Valencia-Cabrera, L., Pérez-Jiménez, M.J.: Generating Diophantine sets by Virus Machines. In: Gong, M., Pan, L., Song, T., Tang, K., Zhang, X. (eds.) Bio-Inspired Computing—Theories and Applications, Communications in Computer and Information Science, vol. 562, pp. 331–341. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49014-3_30
9. Romero-Jiménez, Á., Valencia-Cabrera, L., Riscos-Núñez, A., Pérez-Jiménez, M.J.: Computing partial recursive functions by Virus Machines. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 9504, pp. 353–368. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28475-0_24
10. Valencia-Cabrera, L., Pérez-Jiménez, M.J., Chen, X., Wang, B., Zeng, X.: Basic Virus Machines. In: Sempere, J.M., Zandron, C. (eds.) Proceedings of the 16th International Conference on Membrane Computing (CMC 2015). pp. 323–342. Valencia, Spain (2015)

# The power of Virus Machines: generating strings

Antonio Ramírez-de-Arellano-Marrero[1,2,3][0000−0002−4666−2494], David Orellana-Martín[2,3][0000−0002−2892−6775], and Francis George C. Cabarle[4][0000−0002−5006−6310]

[1] Research Institute for Artificial Intelligence "Mihai Draganescu", 050711 Bucharest, Romania antonio@racai.ro
[2] Research Group on Natural Computing, Dept. of Computer Sciences and Artificial Intelligence, University of Seville, 41012, Seville, Spain
[3] SCORE Laboratory, I3US, University of Seville, 41012 Seville, Spain.
[4] Dept. of Computer Science, University of the Philippines Diliman, Quezon City, 1101 Philippines

The paper explores the computational potential of virus machines, a recent computational model inspired by biological viruses. It introduces the *generating strings mode*, which allows virus machines to produce sets of strings. The study includes initial results that show that every regular language can be generated by a bounded virus machine, contributing to a better understanding of virus machines' capabilities.

**Keywords:** Virus Machines · Regular Languages · Computability.

## 1 Introduction

Virus machines [6] (VM) are an unconventional computing paradigm inspired by the propagation of biological viruses. Here, viruses are stored in hosts that work as memory units, these hosts are connected by directed arcs (channels) with weight which represents the replication rate of each host. How viruses are propagated and replicated is controlled by instruction units that are attached to those channels and open/close them.

Briefly, a VM is composed of three graphs: a directed *host graph* with hosts as nodes for storage of viruses, and *channels* are edges between hosts for virus replication; a directed *instruction graph* with instructions to activate as nodes, and edges to indicate the next instruction(s); lastly, a nondirected *instruction-channel graph* connecting an instruction node to an edge (channel) in the host graph to allow replication. Precise syntax and semantics of VMs are in Definition 2 and Section 3.1. For further specific definitions, notation, and results we refer the interested reader to [2,10].

Recently, the computational power of VM has been studied in the generation mode of natural numbers sets when some of the ingredients that concern the network are (un)bounded [2]. In this work, we propose the idea of not generating sets of natural numbers but now generating languages, that is, sets of strings, using VM. Lastly, examples and a standard form are presented. The present work summarises and extends the preliminary work in [3]. The study of the power for string generation of models is a common line of investigation in informatics, see

e.g. [8,9] early in the 20th century. In more recent and related models, such line is also quite active, e.g. [5,13] for spiking neural P systems and [14] for numerical P systems.

The study of the string generation of VMs contributes to a deeper understanding of their computing power, besides computing numbers. Unlike sets of numbers, sets of strings require a specific order of symbols from a given alphabet. Besides, strings are often associated with "text matching", with a successful theory built around their classifications such as the Chomsky hierarchy, see e.g. [7,11].

## 2 Preliminaries

In this section, to prove our first result let us recall the definition of a regular language. For definitions of regular sets and related notions, the reader can consult monographs and references such as [7,11] and [12]. Let us first remind what is a regular expression:

**Definition 1.** *Given the finite alphabet $\Sigma$, a regular expression $A$ can be obtained by:*

1. $\forall a \in \Sigma$ are regular expressions;
2. empty string $e$ and empty set $\emptyset$ are regular expressions;
3. if $E_1$ and $E_2$ are regular expressions, then their union $E_1 \cup E_2$, their concatenation $E_1E_2$, and star $E_1^*$ are regular expressions.

We denote by $REG$, $CF$, and $CS$ the class of regular, context-free, and context-sensitive sets, respectively, recalling that regular expressions characterise $REG$, see e.g. [7,11].

Let us recall the syntax of virus machines, here a brief overview is presented, for a more detailed presentation, see [2].

**Definition 2.** *A virus machine (VM) of degree $(p, q)$ with $p, q \geq 1$ is defined as:*

$$\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$$

*where:*

- $\Gamma = \{v\}$ *is the Singleton alphabet, that is there is only one type virus;*
- $H = \{h_1, \ldots, h_p\}$ *is the ordered set of hosts;*
- $I = \{i_1, \ldots, i_q\}$ *is the ordered set of instructions, being $i_1$ the initial instruction;*
- $D_H = (H \cup \{h_{out}\}, E_H, w_H)$ *is a weighted directed graph called the host graph, the arcs are denoted as channels between the hosts with $E_H \subseteq H \times H \cup \{h_{out}\}$, and $w_H$ is a map from $E_H$ to $\mathbb{N}$;*
- $D_I = (I, E_I, w_I)$ *is a weighted directed graph called the instruction graph, here the out-degree of each node is at most 2 and $w_I$ is a map from $E_I$ to $\{1, 2\}$.*

- $G_C = (I \cup E_H, E_C)$ *is an undirected bipartite graph called the instruction-channel graph, where the partition is* $\{I, E_H\}$*. Each instruction* $I$ *is attached to at most one channel.*
- $n_1, \ldots, n_p \in \mathbb{N}$ *are the initial number of viruses at each host* $h_1, \ldots, h_p$*, respectively;*
- $h_{out}$ *represents the output region, it can be either* $h_{out} \in H$ *or* $h_{out} \notin H$*. For the present work,* $h_{out} \notin H$ *and it will be denoted as* $h_0 = h_{out}$*.*

Regarding the semantics, a configuration of the system at an instant $t \geq 0$ is denoted as $C_t = (a_{1,t}, \ldots, a_{p,t}, u_t, a_{0,t})$, where $a_{j,t} \in \mathbb{N}$ is the number of viruses at the host $h_j$ at instant $t$, and $u_t \in I \cup \{\#\}$ is the instruction that will be activated in the next transition step. If $u_t = \#$, then $C_t$ is a *halting configuration*. The *initial configuration* is $C_0 = (n_1, \ldots, n_p, i_1, 0)$.

Let us see how a transition step from $C_t$ to $C_{t+1}$ is defined. The instruction that will be activated is $u_t$ if $u_t \in I$, otherwise $C_t$ is a halting configuration. Let us suppose that $u_t \in I$ and that it is attached to the channel $(h_j, h_{j'}) \in E_H$ with weight $w_H((h_j, h_{j'})) = w \in \mathbb{N}$, then the channel is *opened* and two possibilities hold:

- If $a_{j,t} > 0$, then there is *virus transmission*, that is, one virus is consumed from $h_j$ and is sent to the host $h_{j'}$ replicated by $w$. The next activated instruction will follow the highest weight path from $u_t$ to just the next instruction on the instruction graph. In case the highest path is not unique, it is chosen nondeterministically. In case there is no possible path, then $u_{t+1} = \#$
- If $a_{j,t} = 0$, then there is no virus transmission and the next instruction follows the least weight path from $u_t$ to just the next instruction. For the other cases, it is analogous to the previous assumption.

A computation of a VM $\Pi$ is a (finite or infinite) sequence of configurations such that: (i) the first term is the initial configuration $C_0$; (b) for each $n \geq 1$, the $n$-th term of the sequence is obtained from the previous term in one transition step; and (c) if the sequence is finite (called *halting computation*) then the last term is a halting configuration.

## 3 Definitions

### 3.1 Generating strings mode

For generating strings, a new semantics must be defined. Previous to the definition, let us fix the following for a given VM:

1. Let $H$ be the host set of the VM;
2. We are working on the finite alphabet $\Sigma$ and the empty symbol is denoted by $e$;
3. The output region is not included in the host graph, that is $h_{out} = h_0$ where $h_0 \notin H$;
4. The number of hosts of the VM is at least $|\Sigma| + 1$;

5. There exists a surjective function $l : H \to \Sigma \cup \{e\}$, that maps each host to a symbol from the alphabet $\Sigma$ or the empty symbol $e$;
6. For simplicity, if for a host $h_j \in H$, we have $l(h_j) = k \in \Sigma \cup \{e\}$, then it will be visually represented as $h_k$.
7. The weight of the channels that are attached to the host $h_{out}$ are equal to 1.

Regarding now the semantics, the output string is read by the following manner.

**Definition 3.** *Let $\Sigma$ be some alphabet. If at a transition step there is at least one virus transmitted from a host $h_j$ with $l(h_j) = x \in \Sigma$ to the environment $h_0$, then the symbol $x \in \Sigma$ repeated $w_H(h_j, h_0)$ times is added to the right of the string. In this way a string $x = x_1x_2 \dots x_k \in \Sigma^*$ is generated by a VM $\Pi$ if and only if there exists at least one halting computation which transmits a symbol $x_i$ at step $i$, for $1 \leq i \leq k$. The union of all such generated strings forms the language of $\Pi$, denoted by $L(\Pi)$.*

*Let $\Sigma$ be an alphabet and $w_0 = e$. For every transition $C_t \Rightarrow C_{t+1}$,*

$$w_{t+1} = \begin{cases} w_t a, \text{ if the transition uses} (h_j, h_0), l(h_j) = a \in \Sigma \\ w_t, \quad \text{otherwise} \end{cases}$$

*Then,*

$$L(\Pi) = \{w_T \mid C_0 \Rightarrow^* C_T \text{ and } C_T \text{ is halting}\},$$

*where $\Rightarrow^*$ is a computation of $\Pi$, $L(\Pi)$ is the language of $\Pi$.*

**Definition 4.** *Let $LVM(p, q, r)$ the family of languages generated by VMs with at most $p$ hosts, $q$ instructions, and at most $r$ viruses at each host during the computation. If any ingredient is unbounded, it will be denoted by $*$.*

### 3.2 Explicit Examples

To clarify the novel semantics, let us see an explicit example. Let $\Sigma = \{a, b\}$ be the alphabet and $\Pi_{ex}$ be the VM of degree $(3, 4)$ visually described in Figure 1. The classical representation is followed, the squares are the hosts, the blue dots the instructions, and the red dashed lines represent the instruction-channel graph. Here $h_1, h_2$, and $h_3$ are labeled as $a, b$ and $e$ respectively.

In the set $L(\Pi_{ex})$, all the possible computations must be studied. The initial configuration is $C_0 = (1, 2, 3, i_1, 0)$, there is a virus sent to the environment from host $h_1$, thus the symbol $a$ is added to the right of the string. From here a non-deterministic decision is taken, going to instruction $i_2$ or $i_3$:

1. If instruction $i_2$ is chosen, then the following configuration is $C_1 = (0, 2, 3, i_2, 1)$, from here a virus is sent to the environment from $h_2$, so the string now is $ab$ and we go back to instruction $i_2$ as the highest weight path is followed. From the configuration $C_2 = (0, 1, 3, i_2, 2)$, we have the same process, the output string is now $ab^2$, and we come back to $i_2$. As the host $h_2$ is empty, the next activated instruction is $i_4$, from which a halting configuration is reached $C_5 = (0, 0, 3, \#, 5)$, with the following string generated $ab^2$.

2. If instruction $i_3$ is chosen, then we have the following configuration $C_1 = (0, 2, 3, i_3, 1)$, from here a virus is transmitted from host $h_3$ to the environment and the following configuration is $C_2 = (0, 2, 2, i_3, 2)$. As the label of $h_3$ is $e$, no symbol is added to the output string. After that, the halting configuration is reached without further virus transmission, having generated the string $a$ with the halting configuration $C_3 = (0, 2, 2, \#, 2)$.

In conclusion, $L(\Pi_{ex}) = \{a, ab^2\}$.

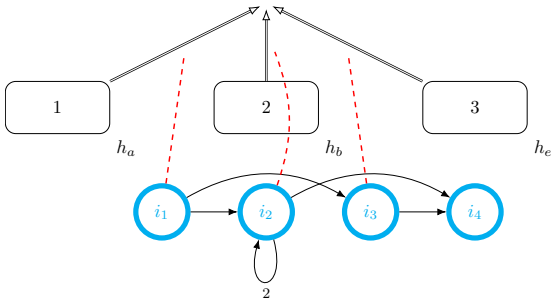



Fig. 1: VM $\Pi_{ex}$ with $L(\Pi_{ex}) = \{a, ab^2\}$,

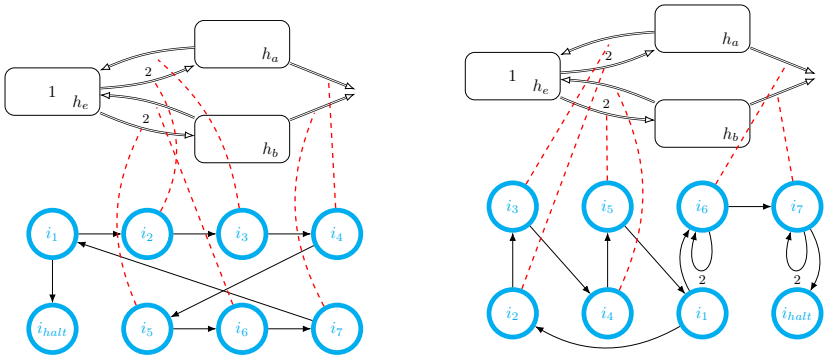
For the next example we use again $\Sigma = \{a, b\}$ and let $\Pi_1$ be the VM visually represented in Figure 2a to generate the language $\{(ab)^i \mid i \geq 0\} = \{ab\}^*$. The specific topology used in $\Pi_1$ and other VMs later is denoted as a standard form in Section 3.3. Note that $L(\Pi_1) \in REG$ since one regular expression for the language is the expression $(ab)^*$.

A technique to formally verify the VM is by proving invariant formulas that highlight the relevant loops of the devices. This technique has been applied in several works [2,4]. Here, not only the formula but also the string that is generating in the computation is included.

For $\Pi_1$, suppose that the generated string is $(ab)^n$ for some $n \geq 0$, then we have the following invariant formula:

$$\varphi(k) \equiv C_{7k} = (0, 0, 1, i_1, 2k), \text{ with string } (ab)^k \text{ for each } 0 \leq k \leq n;$$

For this loop, all instructions except $i_{halt}$ are included, the key is that a non-deterministic decision is made at $i_1$, where we can finish the computation or start a new loop again. As $\varphi(k)$ is true for $1 \leq k \leq n$, then $\varphi(n) \equiv C_{7n} = (0, 0, 1, i_1, 2n)$ with the string $(ab)^n$ is true. Now the non-deterministic decision is to take instruction $i_{halt}$; thus, after 2 transition steps, the generated string is $(ab)^n$ and the halting configuration is $C_{7n+2} = (0, 0, 1, \#, 2n)$.




(a) VM $\Pi_1$ that generates the language $\{(ab)^i \mid i \geq 0\}$.

(b) VM $\Pi_2$ that generates the language $\{a^i b^i \mid i \geq 0\}$.

Fig. 2: VMs $\Pi_1$ and $\Pi_2$ that compute a regular language and a typical nonregular context-free language, respectively.

### 3.3 Standard form

Having shown the novel semantics, for the characterizations of family of languages, it can be convenient to initially fix some characteristics of the VMs that will be used, they will be called *standard form*.

**Definition 5.** *Let $\Sigma = \{a_1, \ldots, a_p\}$ be an alphabet of $p > 0$ symbols, then A VM $\Pi$ in generating mode is in standard form over $\Sigma$ if it verifies the following:*

1. *The initial instruction is $i_1$ and all the halting computations must come from the same instruction that will be called $i_{halt}$.*
2. *The number of hosts is $p+1$, each $h_j$ associated with the symbol $a_j$ for each $1 \leq j \leq p$, and $h_{p+1}$ is associated with the empty string and denoted by $h_e$.*
3. *The initial amount of viruses on each host is 0 except for $h_e$ that initially has one virus. The same is true for the halting configuration.*
4. *The host graph must be as shown in Figure 3.*

This restriction on the host graph is not coincidental; if we look at the work in [4], we can see that the authors designed a small universal VM whose host graph satisfies the aforementioned restrictions; therefore, it can be interpreted as the universal VM being in standard form.

Next we provide the start of our main results, that of generating regular languages.

## 4 Results

### 4.1 Regular Languages

Next, we consider an infinity of strings, in particular the class *REG* from Section 2.

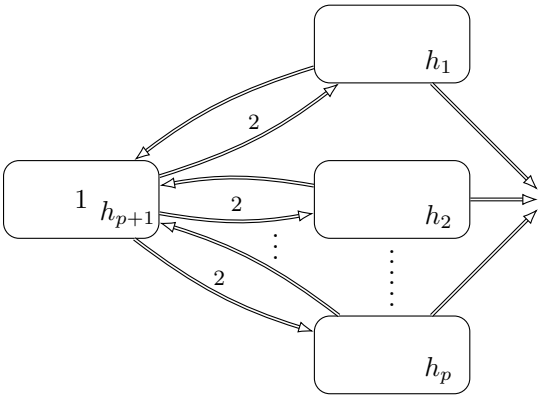



Fig. 3: The host graph of VM in standard form over $\Sigma$. Note that $p$ is the amount of symbols in $\Sigma$

**Proposition 1.** *Let a natural number $n > 0$ and $REG_n$ be the set of regular languages over an alphabet $\Sigma$ of $n$ symbols, then $REG_n \subseteq LVM(n+1, *, 2)$.*

*Proof.* The idea of the proof is to construct a VM in generating strings mode that simulates each property that characterise $REG_n$.

- For each symbol $a \in \Sigma$, let $\Pi_a$ be the VM that generates the set of strings $\{a\}$ visually described in Figure 4. The idea is to generate one extra virus to host $h_a$ and the send it to the environment, note that the halting configuration $h_e$ remains with only one virus and $h_a$ is empty. Let us see the explicit computation:

$$C_0 = (0, 1, i_1, 0)$$
$$C_1 = (2, 0, i_2, 0)$$
$$C_2 = (1, 1, i_3, 0)$$
$$C_3 = (0, 1, \#, 1)$$

- Let $e$ be the empty string, and $\Pi_e, \Pi_\emptyset$ be the VM that generates the empty string and the empty set respectively and visually represented in Figure 5. The proof is almost trivial, as the first device halts in step 1, sending no viruses to the environment, thus the empty string has been generated, as it is the only possible computation, the set $\{e\}$ has been generated. For the second one, no halting computation can be reached, then no string can be generated and the language generated is the $\emptyset$.
- Let $E_1, E_2 \in REG_n$, then let us suppose that there exists two VMs $\Pi_1, \Pi_2$ from the family of VM in the proposition that generates the string $E_1$ and $E_2$ respectively, then there exists a VM $\Pi_{E_1 \cup E_2}$ that generates $E_1 \cup E_2$ from the family of VM visually described in Figure 6. Here the initial instruction is $i_{1'}$ and the halting instruction is $i_{halt'}$.

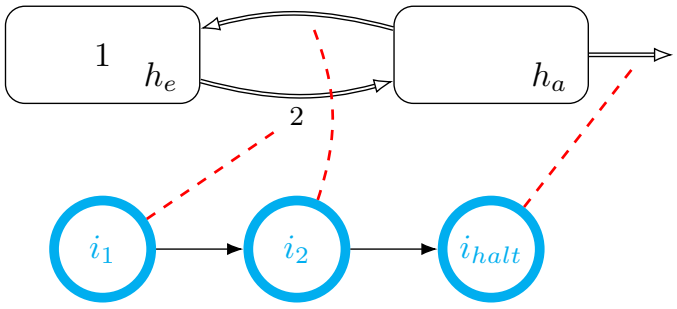
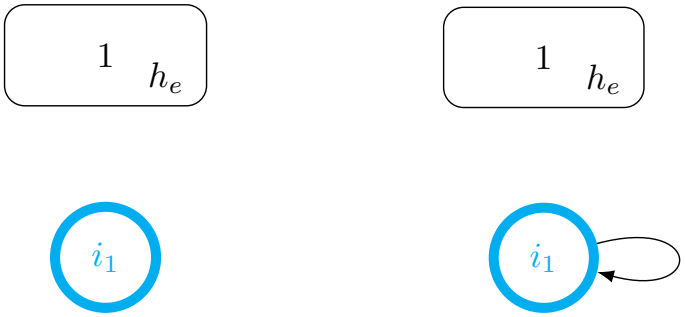



Fig. 4: VM $\Pi_a$ generating $\{a\}$.




(a) VM $\Pi_e$ generating the empty string. (b) VM $\Pi_\emptyset$ generating the empty set.

Fig. 5: VMs generating the empty string and the empty set.

Initially, from instruction $i_{1'}$, a nondeterministic decision is taken between the instruction $i_1$ from $\Pi_1$ and instruction $i_1$ form $\Pi_2$, and after all the possible computations from both devices can be taken, thus generating $E_1 \cup E_2$.

– Let $E_1, E_2 \in REG_n$, then let us suppose that there exists two VMs $\Pi_1, \Pi_2$ from the family of VM in the proposition that generates the string $E_1$ and $E_2$ respectively, then $E_1E_2$ can be generated by the VM $\Pi_{E_1E_2}$ visually described in Figure 7. Here the initial instruction is the instruction $i_1$ from $\Pi_1$, and the halting instruction is the instruction $i_{halt}$ from $\Pi_2$. Moreover, the instruction $i_{halt}$ from $\Pi_1$ is attached to $i_{1'}$, and this one is attached to the instruction $i_1$ from $\Pi_2$. About the host graph, it is the same for both $\Pi_1$ and $\Pi_2$, so $\Pi_{E_1E_2}$ has the same.

  Let the string $s_1 \in E_1$, then there exists a finite computation from $\Pi_1$ that generates $s_1$, thus at $\Pi_{E_1E_2}$ there exists a step finite sequence of configurations that reaches $i_{1'}$ and then the $i_1$ from $\Pi_2$. Here, it is important to note that we have the standard form, thus every host is empty except $h_e$ that has only one virus, that is the initial configuration of $\Pi_2$ except for the output host that does not affect in the computation.

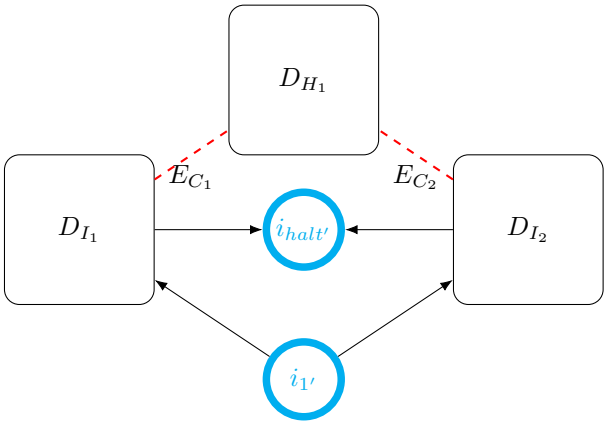



Fig. 6: VM $\Pi_{E_1 \cup E_2}$ generating $E_1 \cup E_2$.

After that, let the string $s_2 \in E_2$, then there exists a computation from $\Pi_2$ that generates $s_2$. Taking that previous computation of $\Pi_{E_1 E_2}$, it can be continued by the computation that generates $s_2$, thus generating the string $s_1 s_2$ for any $s_1 \in E_1, s_2 \in E_2$.

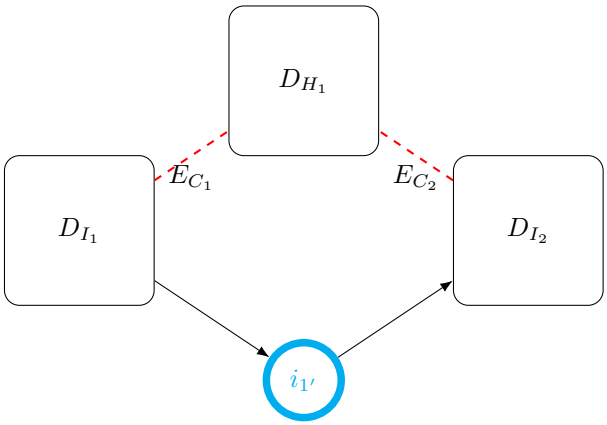



Fig. 7: VM $\Pi_{E_1 E_2}$ generating $E_1 E_2$.

– Let $E \in REG_n$, then let us suppose that there exists a VM $\Pi_1$ from the family of VM in the proposition that generates the string $E$, then $\Pi_*$ is the VM that generates $E^*$ visually described in Figure 8 and explicitly constructed as follows.

$D'_I = (I', E'_I, w'_I)$, where $I' = I \cup \{i_{start}, i_{halt'}\}$, $E'_I = \{(i_{start}, i_1), (i_{start}, i_{halt'}), (i_{halt}, i_1), (i_{halt}, i_{halt'})\} \cup E_I$, with $w'_I(i_{halt}, i_1) = w_I(i_{halt}, i_{halt'}) = w_I(i_{start}, i_1) = w_I(i_{start}, i_{halt'}) = 1$, otherwise $w'_I(i_j, i_k) = w_I(i_j, i_k)$.

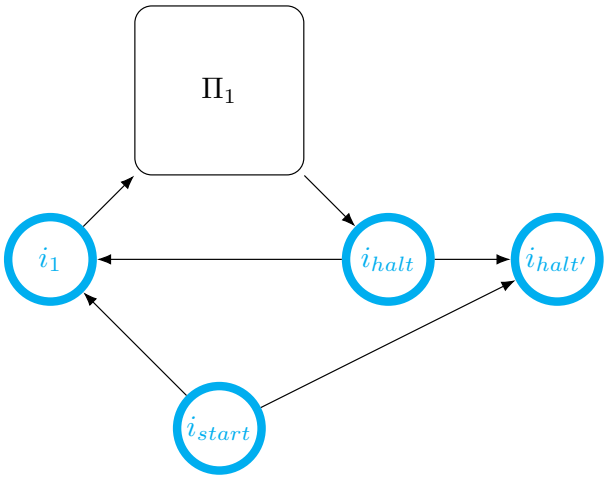



Fig. 8: VM $\Pi_*$ generating $E^*$.

Recall that for the proof, only the hosts and channels from the standard form are necessary, the viruses are bounded by 2 viruses at each host, the only ingredient that increases is the number of instructions, that is unbounded.

On the other hand, what remains in the proof is that, for each VM of the family mentioned, there is no string generated out of a regular language, for that all the modules presented in the proof must be reviewed.we will focus on the concatenation module already presented Figure 7 with the same notation.

For the modules in Figures 4 and 5, the strings that they can generate are regular languages. Next modules, let us fix the notation: $E_1$ is the language generated by $\Pi_1$ and $E_2$ the language generated by $\Pi_2$.

For the union module represented in Figure 6, let $w$ an arbitrary string generated by the union module. From instruction $i_{1'}$ we can non-deterministically go to $\Pi_1$ or $\Pi_2$, thus the string $w$ will be either $w_1 \in E_1$ or $w_2 \in E_2$. Hence, $w \in E_1 \cup E_2$, and therefore, $L(\Pi_{E_1 \cup E_2}) \subseteq E_1 \cup E_2$.

For the concatenation module represented in Figure 7. Let $w$ an arbitrary string generated by the concatenation module, initially $\Pi_1$ generates a string $w_1 \in E_1$, then instruction $i_{1'}$ is activated and then from the VM $\Pi_2$ the string $w_2 \in E_2$ is included to the right of $w_1$, thus $w = w_1 w_2 \in E_1 E_2$.

Lastly, for the star module represented in Figure 8, let $w$ the string generated by the VM. Initially form instruction $i_{start}$ a non-deterministic decision has to be chosen, if the chosen one is the instruction $i_{halt'}$ then $w$ has no symbols. if instruction $i_1$ is chosen, the string $w_1 \in E_1$ is generated and the instruction

$i_{halt}$ is reached, from here to option arises, if instruction $i_{halt'}$ is chosen, then the string generated is $w = w_1 \in E_1$. Otherwise instruction $i_1$ is chosen again and a new word $w_2 \in E_1$ is generated. Repeating this process $k$ times yields $w = w_1 w_2 \dots w_k$, with $w_j \in E_1$ for every $1 \leq j \leq k$. Hence, $w \in E_1^*$.

**Corollary 1.** $REG \subseteq LVM(*, *, 2)$

*Proof.* Since any regular language $A$ belongs to some $REG_n$, it follows that $REG \subseteq LVM(*, *, 2)$.

### 4.2 Beyond *REG*

Consider now the VM $\Pi_2$ from Figure 2b. Suppose that the generated string is $a^n b^n$ for some $n \geq 0$, then we have the following invariant formula:

$$\varphi'(k) \equiv C_{5k} = (k, k, 1, i_1, 0), \text{ for each } 0 \leq k \leq n;$$

Now, the loop involves instructions from $i_1$ to $i_5$. $\varphi'(n)$ is true so the configuration $C_{5n} = (n, n, 1, i_1, 0)$ is reached. From here, the non-deterministic decision is to go to instruction $i_6$, now all viruses from host $h_a$ are sent to the environment, generating the string $a^n$ with the configuration $C_{6n+1} = (0, n, 1, i_6, n)$. From here it is analogous with instruction $i_7$ and the host $h_b$, after which the halting configuration $C_{7n+4} = (0, 0, 1, \#, 2n)$ is reached with the generated string $a^n b^n$.

*Note 1.* It is interesting to note that the VM $\Pi_2$ in Figure 2b can be extended in a "natural" way as $\Pi_3$ to have a new host $h_c$ such that $L(\Pi_3) = \{a^i b^i c^i \mid i \geq 0\}$. Moreover, it can be extended analogously to $\Pi_n$ such that $L(\Pi_n) = \{a_1^i \cdots a_n^i \mid i \geq 0\}$.

**Theorem 1.** *The class $LVM(*, *, *)$ fulfills the following statements:*

1. $LVM(*, *, *) \cap (CF \setminus REG) \neq \emptyset$,
2. $LVM(*, *, *) \cap (CS \setminus CF) \neq \emptyset$.

*Proof.* For the first set, the proof is based on the construction of $\Pi_2$ since $\{a^i b^i \mid i \geq 0\}$ is in $CF$ but not in $REG$. For the second set, the proof is based on the construction of $\Pi_3$ since $\{a^i b^i c^i \mid i \geq 0\}$ is in $CS$ but not in $CF$.

## 5 Conclusion

In this work a new semantics is presented in VM for generating strings, creating a new scope in the computational power study of these devices.

With the novel standard form, new possibilities arise for making characterisations of a class of languages when some ingredients in the VM are bounded. For instance, including a characterisation of finite, regular, and other language classes such as those from the Chomsky hierarchy or other "interesting" languages outside the usual Chomsky classes.

Another possible scope is taking the results from [1] with characterisations of families of natural number sets. Those results can be taken and provide morphisms to convert sets of numbers into sets of strings. From [1] specific bounds for $SLIN$ or the semilinear sets of numbers are given, noting that $SLIN$ is the length sets of both regular and context-free sets. Results from our present work and in [1] can be combined to give new results or better bounds. The results in the present work use the standard form from Section 3.3, so a less or more strict form seems interesting. Consideration of the accepting mode is also interesting: as with accepting numbers, strings can be the input to certain hosts with the VM halting (perhaps with a specific output symbol) if the input is accepted.

**Acknowledgments.** This work is part of Project PID2025-174204NB-I00, funded by the Ministry of Science, Innovation and Universities (MICIU), the State Research Agency (AEI), DOI 10.13039/501100011033, and the European Regional Development Fund (ERDF), European Union. The work of Antonio Ramírez de Arellano Marrero was supported by the Marie Sklodowska-Curie Actions grant 101206770 - VM-AI. F.G.C. Cabarle acknowledges support from the 2024–2026 Scientific Productivity System of the University of the Philippines (UP), the San Francisco and Mely and Rick Rey Foundation Professorial Chair of the UP Alumni Association, and RLC from the UP Diliman OVCRD.

Last but not least, we thank and dedicate the present work to Mario J. Pérez-Jiménez. Thank you very much Mario for showing us the way in scientific investigations: your generosity, dedication, and persistence leave a lasting mark not only in science but also in our lives and others. This is the way.

## References

1. Ramírez-de Arellano, A., Cabarle, F.G.C., Orellana-Martín, D., Pérez-Jiménez, M.J.: Normal forms in virus machines (2025), https://arxiv.org/abs/2409.03327
2. Ramírez-de Arellano, A., Cabarle, F.G.C., Orellana-Martín, D., Pérez-Jiménez, M.J.: On the normal forms and computational power of virus machines. Theoretical Computer Science p. 116139 (2026). https://doi.org/10.1016/j.tcs.2026.116139
3. Ramírez-de Arellano, A., Orellana-Martín, D., Cabarle, F.G.C.: Virus machines and strings. 21st Brainstorming Week on Membrane Computing, 22–24 January 2025, Sevilla, Spain pp. 93–105 (2025)
4. Ramírez-de Arellano, A., Orellana-Martín, D., Pérez-Jiménez, M.J.: Using virus machines to compute pairing functions. International Journal of Neural Systems **33**(05), 2350023 (2023). https://doi.org/10.1142/S0129065723500235
5. Chen, H., Freund, R., Ionescu, M., Păun, G., Pérez-Jiménez, M.J.: On string languages generated by spiking neural P systems. Fundamenta Informaticae **75**(1-4), 141–162 (2007). https://doi.org/10.3233/FI-2016-1372
6. Chen, X., Pérez-Jiménez, M.J., Valencia-Cabrera, L., Wang, B., Zeng, X.: Computing with viruses. Theoretical Computer Science **623**, 146–159 (2016). https://doi.org/10.1016/j.tcs.2015.12.006
7. Hopcroft, J.E., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley series in computer science, Addison-Wesley, Reading, Mass. (1979)

8. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Automata studies, Princeton, N. J., vol. 34, pp. 3–41. Princeton University Press. Annals of mathematics studies (1956)
9. Power, J.F.: An early completion algorithm: Thue's 1914 paper on the transformation of symbol sequences. In: Beckmann, A., Csuhaj-Varjú, E., Meer, K. (eds.) Language, Life, Limits. pp. 343–346. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08019-2_35
10. Ramírez-de-Arellano, A., Orellana-Martín, D., Pérez-Jiménez, M.J.: Generating, computing and recognizing with virus machines. Theoretical Computer Science **972**, 114077 (07 2023). https://doi.org/10.1016/j.tcs.2023.114077
11. Salomaa, A.: Formal languages. Academic Press Professional, Inc., USA (1987)
12. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages: Volume 1 Word, Language, Grammar, pp. 41–110. Springer Berlin Heidelberg, Berlin, Heidelberg (1997), 10.1007/978-3-642-59136-5_2
13. Zhang, L., Xu, F., Ramanujan, A., Subramanian, K.G.: Control languages accepted by labeled spiking neural P systems with rules on synapses. Theoretical Computer Science **893**, 60–71 (2021). https://doi.org/10.1016/j.tcs.2021.06.027
14. Zhang, Z., Wu, T., Pan, L.: On string languages generated by sequential numerical P systems. Fundamenta Informaticae **145**(4), 485–509 (2016). https://doi.org/10.3233/FI-2016-1372

# An assembly P system which generates finite approximations of the Hilbert curve

Taishin Y. Nishida[0000−0003−2253−8306]

A free researcher,
Fellow of Japan Accreditation Board for Engineering Education
(formerly Department of Information Science, Toyama Prefectural University)
nishida.yasunobu.g79@kyoto-u.jp

**Abstract.** An assembly P (AP) system generating approximations of the Hilbert curve is proposed. An AP system is a variant of tissue P systems in which rules specify connections of cells and extra cellular matrix (ECM) objects. The proposed AP system first produces ECM objects and put them at the vertices in the $(n + 1)$-st approximated Hilbert curve using the $n$-th approximation. Then cells divide and form the edges of the $(n + 1)$-st approximation. Cells in the proposed system play the symbols of chain code, which represent line segments of unit length of right, left, up, or down directions. It is proved that the proposed system satisfies the recursive expression over the $n$-th and $(n + 1)$-st approximated Hilbert curves.

**Keywords:** Assembly P systems · Space filling curves · Hilbert curve · D0L system.

## 1 Introduction

The author has proposed, as a variant of P systems, assembly P (AP) systems, inspired from the phenomena that living cells form proper shapes and perform appropriate functions according to the environmental stimuli [5]. Cells in an AP system divide and differentiate such that they assemble a shape and perform a function. The shape and the function are suited to the environment for the cells, called extra-cellular matrix (ECM) objects. Two systems have been introduced in [5] as concrete examples of AP systems: One solves a given instance of the Hamiltonian cycle problem or the satisfiability problem for a Boolean expression and the other simulates a given linear bounded automaton. In [5] the ECM objects do not change during the development of the AP system and hence the size of the constructed shape is bounded by the size of the input (the initial ECM objects).[1]

As the next step of AP systems, here we will propose systems in which cells make matrix objects and then they attach to the new objects and assemble

[1] In the proposed AP systems, only cell-ECM object connections are permitted and cell-cell connections are inhibited to avoid forming a chaotic lamp of cells like turmours.

shapes of arbitrary size. We also introduce absolute coordinates to place building blocks or cells and ECM objects at proper positions in the 2-dimensional[2] space while, in [5], rules only define connecting relations between cells and ECM objects and hence cannot determine a unique shape. As an example of the above mentioned systems, we construct an AP system which generates the finite approximations of the Hilbert curve. The Hilbert curve is a space-filling curve discovered by D. Hilbert in 1896 as the second space-filling curve[3]. A large number of investigations on space-filling curves, from mathematical analyses to applications to various fields including computer science, have been done over one century. Those who are interested in the results about the space-filling curves are referred to [1].

Here we focus our interest on generating recursive approximations of the Hilbert curve. A simplified 2-dimensional space-filling curve is defined as a surjective and continuous mapping $f : [0, 1] \to [0, 1]^2$. It is convenient to define a space-filling curve as a limit of recursively defined curves $f_1, f_2, \ldots, f_n, \ldots$. As for the Hilbert curve, the first few curves of the sequence are shown in Figure 1. Most recursive sequences use the chain code expression or the turtle expression. The chain code consists of $r$, $l$, $u$, and $d$ which express right, left, up, and down, respectively, line segments of unit length. The turtle expression uses $\uparrow$, $\turnleft$, and $\turnright$, which are forward, turn left, and turn right, respectively. A recursive expression of the chain code looks like [2]:

$$\begin{aligned} H_1 &= urd, \\ H_n &= (m \circ \lambda^3)(H_{n-1})uH_{n-1}rH_{n-1}d(m \circ \lambda)(H_{n-1}), \end{aligned} \tag{1}$$

where $m$ is the axial symmetric conversion with respect to the vertical line (mirror image), $\lambda^3$ is the clock-wise rotation by $\frac{\pi}{2}$, and $\lambda$ is the counter clock-wise rotation by $\frac{\pi}{2}$. Such recursive expressions are transformed to production rules of grammars [1–4, 6, 8]. But all of the preceding researches use nonterminals and terminals[4]. Since our purpose is to make a growing AP system, we want to build an AP system in which every intermediate step corresponds to an approximation of the Hilbert curve. In other words, we want to make a parallel rewriting system with no discrimination of terminals and nonterminals, such as D0L systems.

Therefore, we first construct a D0L system which generates the finite approximated Hilbert curves iteratively. Then we construct an AP system with absolute coordinate which generates both the chain codes in the D0L system and ECM objects which play the role of the vertices in the approximated Hilbert curves.

## 2 Preliminaries

The readers are assumed to be familiar with formal language theory. Here we use the notations: $\epsilon$ denotes the empty word, $\mathbb{N}$ denotes the set of nonnegative integers, and $\mathbb{R}$ denotes the set of real numbers.

[2] Of course, 3 or higher dimensional systems will be possible.
[3] The first one has been discovered by B. Peano in 1890.
[4] The terminals are the chain code or turtle symbols.

As for theoretical aspects L systems[5], the readers are referred to [7]. The book [6] is a good monograph for the graphical applications of L systems. Here we give a brief definition of D0L systems. A D0L system is a triple $G = (\Sigma, \tau, w)$ where $\Sigma$ is a finite alphabet, $\tau : \Sigma \to \Sigma^*$ is a homomorphism, and $w \in \Sigma^*$ is the axiom. The system $G$ generates a sequence of words $w = \tau^0(w), \tau(w), \tau^2(w), \ldots, \tau^i(w), \ldots$ from $w$. The language generated by $G$, denoted by $L(G)$, is the set of all words in the sequence, or $L(G) = \{\tau^i(w) \mid i \in \mathbb{N}\}$.

An assembly membrane system or an AP system, is a construct [5]

$$\Pi = (O, C, M, s_0, c_0, F, R, \rho)$$

where:

- $O$ is a set of molecular objects.
- $C$ is a set of cell objects.
- $M$ is a set of (extra cellular) matrix objects.
- $s_0 \in C$ is an initial stem cell.
- $c_0 \in C$ is an initial control cell.
- $F \subseteq C$ is a set of accepting cells.
- $R$ is a set of rules.
- $\rho \subseteq R \times R$ is an order relation over $R$ which defines the priority among rules.

A matrix object has a number of connecting modules to assemble cells. We employ the notation $a_i \begin{pmatrix} y \\ \langle x \rangle \end{pmatrix}$ to describe a matrix object with a connecting module where $a_i$ is a matrix object, $x$ is a label of the connecting module, and $y \in \{\lambda, \rightarrow, \leftarrow\} \cup C$ indicates the state of the connecting module. If $y = \lambda$, then no cell connects the module and the module is called *open*. If $y \in C$, then the cell connects the module and the module is called *close*. In case that a cell connects more than one matrix object, then the symbols $\rightarrow$ and $\leftarrow$ are used to indicate that the module is close and the direction of the connecting cell (on the paper), e.g., $v_i \begin{pmatrix} \rightarrow \\ \langle x \rangle \end{pmatrix} c v_j \begin{pmatrix} \leftarrow \\ \langle y \rangle \end{pmatrix}$ expresses that cell $c$ connects $v_i$ at $\langle x \rangle$ and $v_j$ at $\langle y \rangle$. A matrix object with all connecting modules open is denoted by $a(\langle x_1 \rangle \cdots \langle x_n \rangle)$ instead of $a \begin{pmatrix} \lambda & \cdots & \lambda \\ \langle x_1 \rangle & \cdots & \langle x_n \rangle \end{pmatrix}$.

Now we give a formal definition of an assembly of matrix and cell objects.

(a) A cell object is an assembly.
(b) A matrix object is an assembly.
(c) Let $\alpha$ and $\beta$ be assemblies such that there is an open connecting module $\begin{pmatrix} \lambda \\ \langle x \rangle \end{pmatrix}$ in $\alpha$ and there is a cell object $c$ in $\beta$ which can connect $\langle x \rangle$ but does

[5] The letter "L" comes from Aristid Lindenmayer, the originator of this parallel rewriting systems. The letter "0" (number zero) indicates no interactions among letters. The letter "D" means that the system is deterministic.

not connect in $\beta$, Then $\alpha'\beta$ is an assembly in which $\alpha'$ is $\alpha$ with a closed connecting module $\begin{pmatrix} c \\ \langle x \rangle \end{pmatrix}$ or $\begin{pmatrix} \rightarrow \\ \langle x \rangle \end{pmatrix}$[6].

(d) Nothing else is an assembly.

The set of rules $R$ may contain rules over cells and matrices of the form

$$\alpha_1 + \cdots + \alpha_m :\rightarrow \beta_1 + \cdots + \beta_n$$

where $m, n \geq 1$, for every $i \in \{1, \ldots, m\}$ or $j \in \{1, \ldots, n\}$, $\alpha_i$ or $\beta_j$ is an assembly of matrix and cell objects.

In this paper we modify the above AP systems for matrix objects to be produced by cell objects and we add absolute coordinates to matrix objects, which are done in Section 4.

## 3 A D0L system generating approximated Hilbert curves

We use the chain codes $\{r, l, u, d\}$ where $r$, $l$, $u$, or $d$, respectively, represents a right, left, up, or down unit line segment. In the D0L system described below, we also use the capital version $R$, $L$, $U$, and $D$ of $r$, $l$, $u$, and $d$, respectively, to express different types of rules. In addition to the chain codes, symbol $e$ for the 0 vector and its capital version $E$ are used. The D0L system $G = (\Sigma, \tau, e)$ where:

(i) $\Sigma = \{r, l, u, d, R, L, U, D, e, E\}$

(ii) The homomorphism $\tau$ is given by ($\tau(a) = w$ is denoted by $a \rightarrow w$).

$$\begin{array}{ll} r \rightarrow urdR & R \rightarrow LDRr \\ l \rightarrow rulL & L \rightarrow DLUl \\ u \rightarrow rulU & U \rightarrow DLUu\,. \\ d \rightarrow urdD & D \rightarrow LDRd \\ e \rightarrow urdE & E \rightarrow LDRe \end{array}$$

(iii) $e$ is the axiom.

generates the approximated Hilbert curves iteratively.

The first steps look like

$$\begin{aligned} &\text{step } 0 = e, \\ &\text{step } 1 = urdE, \\ &\text{step } 2 = rulU\ urdR\ urdD\ LDRe, \\ &\text{step } 3 = urdR\ rulU\ \ rulL\ \ DLUu\ rulU\ \ urdR\ \ urdD\ \ LDRr \\ &\qquad\quad rulU\ \ urdR\ urdD\ LDRd\ DLUl\ LDRd\ LDRr\ urdE, \end{aligned}$$

[6] Because $c$ is containd in $\beta$ which is placed right of $\alpha'$ (on the paper), the right arrow must be used. The concatenation $\beta\alpha''$ is possible in which $\alpha''$ may have connecting module $\begin{pmatrix} \leftarrow \\ \langle x \rangle \end{pmatrix}$.

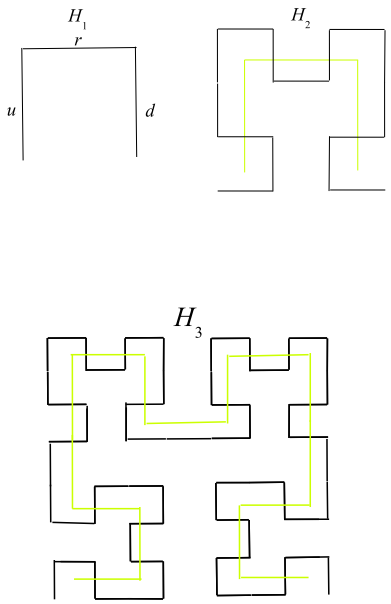



**Fig. 1.** The first three approximations of the Hilbert curve.

$$\begin{aligned}\text{step } 4 = & rulU\ \ urdR\ \ urdD\ \ LDRr\ urdR\ \ rulU\ \ rulL\ \ DLUu\\ & urdR\ \ rulU\ \ rulL\ \ DLUl\ \ LDRd\ DLUl\ \ DLUu\ rulU\\ & urdR\ \ rulU\ \ rulL\ \ DLUu\ rulU\ \ urdR\ \ urdD\ \ LDRr\\ & rulU\ \ urdR\ \ urdD\ \ LDRd\ DLUl\ \ LDRd\ LDRr\ urdR\\ & urdR\ \ rulU\ \ rulL\ \ DLUu\ rulU\ \ urdR\ \ urdD\ \ LDRr\\ & rulU\ \ urdR\ \ urdD\ \ LDRd\ DLUl\ \ LDRd\ LDRr\ urdD\\ & LDRd\ DLUl\ \ DLUu\ rulL\ \ DLUl\ \ LDRd\ LDRr\ urdD\\ & DLUl\ \ LDRd\ LDRr\ urdR\ \ rulU\ \ urdR\ \ urdD\ \ LDRe.\end{aligned}$$

Graphically the words of the chain codes are illustrated in Figure 1.

Since the upper and lower letters in the chain code have the same graphical meanings, we define the coding $\delta$ over $\Sigma$ by $\delta(rludeRLUDE) = rluderlude$, which satisfies that $\delta(x) = \delta(y)$ implies that $x$ and $y$ have the same graphical meanings for every $x, y \in \Sigma^*$. Now we have the next theorem.

**Theorem 1.** *Let $H_0$ be the empty word and let $H_n$ be the $n$-th approximation by the recurrent equation (1) for every $n \geq 1$. Then for every $n$*

$$\tau^n(H_0 e) = \begin{cases} H'_n e & n \text{ is even} \\ H'_n E & n \text{ is odd} \end{cases}$$

*and $\delta(H'_n) = H_n$.*

The proof of Theorem 1 will be found in Appendix.

## 4 AP systems with absolute coordinates and rule of increasing ECM objects

In L system, there are no intrinsic connections between symbols in a string and meanings in the rendering system. On the other hand, assembly P systems can express geometric placements of cell and matrix objects such that these objects form a desired shape.

We define an AP system with absolute coordinates. Here we define AP system over 2-dimensional Cartesian space. Let $\mathit{\Pi} = (O, C, M, s_0, c_0, m_0, R)$[7] be an AP system with 2-dimensional coordinate where $O, C, M, s_0$, and $c_0$ are the same as an AP system without coordinate, $m_0$ is the initial assembly of matrix and cell objects, and $R$ is a set of rules with coordinate.

A matrix object with absolute coordinate is expressed by

$$\begin{matrix} h \\ [x, y] \end{matrix} \begin{pmatrix} \alpha_1 & \cdots & \alpha_n \\ \langle c_1 \rangle & \cdots & \langle c_n \rangle \end{pmatrix}$$

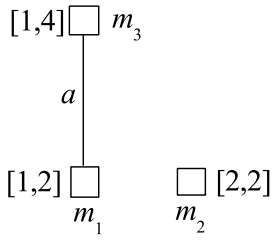
which means that a matrix object $h$ has connecting modules $\langle c_1 \rangle$, ..., $\langle c_n \rangle$ with states $\alpha_1, \ldots, \alpha_n$, respectively, and it locates at $(x, y) \in \mathbb{R}^2$. Every matrix object in a rule has relative coordinates, which is expressed by

$$\begin{matrix} h \\ (p, q) \end{matrix} \begin{pmatrix} \alpha_1 & \cdots & \alpha_n \\ \langle c_1 \rangle & \cdots & \langle c_n \rangle \end{pmatrix}$$

where $p$ and $q$ are linear expression of some variables and coefficients from $\mathbb{R}$ and $(p, q)$ represents relative displacements among matrix objects in the rule. Let $x_1 \ldots, x_n$ be variables in the left-hand side of a rule $r$ and let $\sigma : x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n$ be an assignment for the variables with $a_1, \ldots, a_n \in \mathbb{R}$. Then the objects in the left-hand side get concrete values or absolute coordinates by the assignment. Let $\sigma(r)$ denotes the left-hand side with the absolute coordinates. The position of $\sigma(r)$ can be shifted to $\pi(\sigma(r))$ by a parallel displacement $\pi$.

A rule $r$ can be applicable to a configuration $c$ if there is a sub-assembly $c'$ in $c$, a parallel displacement $\pi$, and an assignment $\sigma$ for the variables appearing in the left-hand side of $r$ such that $\pi(\sigma(r))$ coincides with $c'$ in both positions and matrix and cell objects including the connecting modules. Then $c'$ is replaced with the right-hand side of $r$ where all matrix objects in the right-hand side are given absolute coordinates by applying $\pi$ and $\sigma$. For example let us consider the configuration

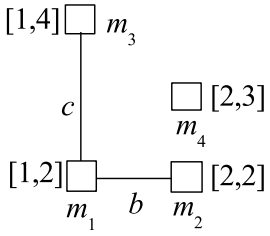
[7] Since we want to generate an infinite sequence of approximated Hilbert curves, the set of accepting cells is omitted. The priority relations are also omitted because there are no needs for priority. On the other hand, we do not consider any input to $\mathit{\Pi}$, then the initial matrix and cell objects $m_0$ is added to the definition of the system.




and the rule

$$\begin{matrix} m_1 \\ (0,0) \end{matrix} \begin{pmatrix} \rightarrow & \lambda \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} a \begin{matrix} m_3 \\ (0,y) \end{matrix} \begin{pmatrix} \lambda & \leftarrow \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} + \begin{matrix} m_2 \\ (x,0) \end{matrix} \begin{pmatrix} \lambda & \lambda \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} \rightarrow$$
$$\begin{matrix} m_2 \\ (x,0) \end{matrix} \begin{pmatrix} \rightarrow & \lambda \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} b \begin{matrix} m_1 \\ (0,0) \end{matrix} \begin{pmatrix} \rightarrow & \leftarrow \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} c \begin{matrix} m_3 \\ (0,y) \end{matrix} \begin{pmatrix} \lambda & \leftarrow \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} +$$
$$\begin{matrix} m_4 \\ (x,0.5y) \end{matrix} \begin{pmatrix} \lambda & \lambda \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix}.$$

The left-hand side coincides with the configuration with displacement $(1,2)$ and assignment $x = 1$ and $y = 2$. Now the configuration




is produced. Here cell object $a$ divides to $b$ and $c$ and a new matrix object $m_4$ is created at position $(1,2) + (1, 0.5 \times 2) = (2,3)$.

Next we give an assembly P system $\Pi = (O, C, M, c_0, s_0, m_0, R)$ which realizes the D0L system generating the approximated Hilbert curves. Let $\Sigma = \{r, l, u, d, R, L, U, D, e, E\}$ and $\Sigma' = \{x' \mid x \in \Sigma\}$. The set of cell objects is given by $C = \Sigma \cup \Sigma' \cup \{s_0\}$. The AP system has no control cells and molecular objects, i.e., $O = \emptyset$ and $c_0 = \epsilon$.[8] The AP system has one type of matrix object $h\langle c_1 \rangle \langle c_2 \rangle$, i.e., $M = \{h\}$ and $m_0 = \begin{matrix} h \\ [0,0] \end{matrix} \begin{pmatrix} s_0 & \lambda \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix}$. The rule for $s_0$ is

$$rs_0 : \begin{matrix} h \\ (0,0) \end{matrix} \begin{pmatrix} s_0 & \lambda \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} \rightarrow \begin{matrix} h \\ (0,0) \end{matrix} \begin{pmatrix} \lambda & \rightarrow \\ \langle c_1 \rangle & \langle c_2 \rangle \end{pmatrix} u$$

[8] Since $\epsilon \notin C$, $c_0 = \epsilon$ is incorrect in the regorous formalism. To avoid this $c_0 = \epsilon$ should be interpreted as an abbreviation of $c_0 \in C$ and $c_0 \rightarrow \epsilon$ is the only rule for $c_0$.

$$\overset{h}{(0,1)}\begin{pmatrix}\leftarrow & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} r \overset{h}{(1,1)}\begin{pmatrix}\leftarrow & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} d \overset{h}{(1,0)}\begin{pmatrix}\leftarrow & E\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix}.$$

The rule sets the first curve and the unit length. Figure 2 shows graphical representation of rule $rs_0$.

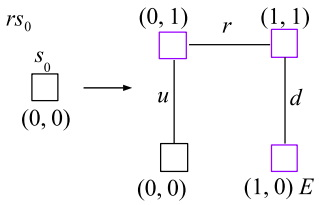



**Fig. 2.** The rule of the initial stem cell $s_0$. This rule makes the first step of the Hilbert curve by creating matrix objects at $(0,1)$, $(1,1)$, and $(1,0)$.

Cell objects from $\Sigma$ create new matrix objects by the rules $rm[x]$ with $x \in \Sigma$. Cell objects from $\Sigma'$ divide and create new cell objects by the rules $rc[x]$ with $x \in \Sigma'$. For cell $r$, $rm[r]$ and $rc[r']$ are

$$rm[r]: \overset{h}{(0,0)}\begin{pmatrix}* & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} r \overset{h}{(x,0)}\begin{pmatrix}\leftarrow & *\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} \rightarrow$$
$$\overset{h}{(0,0)}\begin{pmatrix}* & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} r' \overset{h}{(x,0)}\begin{pmatrix}\rightarrow & *\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} + \overset{h}{(0.5x,0)}\begin{pmatrix}\lambda & \lambda\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} +$$
$$\overset{h}{(0.5x,0.5x)}\begin{pmatrix}\lambda & \lambda\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} + \overset{h}{(0,0.5x)}\begin{pmatrix}\lambda & \lambda\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix}$$

and

$$rc[r']: \overset{h}{(0,0)}\begin{pmatrix}* & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} r' \overset{h}{(x,0)}\begin{pmatrix}\rightarrow & *\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} + \overset{h}{(0.5x,0)}\begin{pmatrix}\lambda & \lambda\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} +$$
$$\overset{h}{(0.5x,0.5x)}\begin{pmatrix}\lambda & \lambda\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} + \overset{h}{(0,0.5x)}\begin{pmatrix}\lambda & \lambda\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} \rightarrow$$
$$\overset{h}{(0,0)}\begin{pmatrix}* & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} u \overset{h}{(0,0.5x)}\begin{pmatrix}\leftarrow & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} r \overset{h}{(0.5x,0.5x)}\begin{pmatrix}\leftarrow & \rightarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} d$$
$$\overset{h}{(0.5x,0)}\begin{pmatrix}\rightarrow & \leftarrow\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix} R \overset{h}{(x,0)}\begin{pmatrix}\leftarrow & *\\ \langle c_1\rangle & \langle c_2\rangle\end{pmatrix}$$

where $*$ stands for $\leftarrow$, $\rightarrow$, or $\lambda$. Rule $rm[r]$ makes 3 new matrix objects at east, north east, and north of the original matrix object to which the start point of $r$ connects. The original and newly created matrix objects form the 4 vertices of a square whose edges have the length of one half of the length of $r$. In rule $rc[r']$ cell $r'$ divides into 4 cells $u, r, d, R$ and they connect the new matrix objects. The rules are graphically represented in Figure 3.

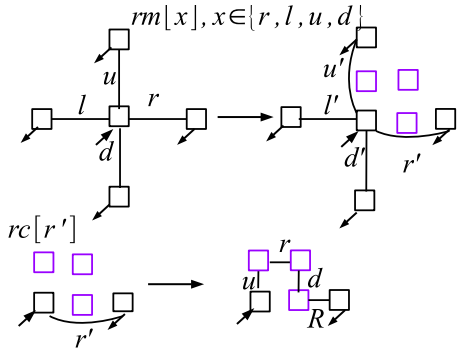



**Fig. 3.** Graphical representations of rules $rm[x]$ $x \in \{r, l, u, d\}$ and $rc[r']$. All four rules $rm[x]$ $x \in \{r, l, u, d\}$ produce three matrix objects at the same place (magenta squares). $rc[r']$ makes cell objects which connect the new matrix objects. The short slanted arrows indicate the start and end points of the chain code

Because graphical representations are easy to understand, we express the other rules graphically. Rules $rm[l]$, $rm[u]$, and $rm[d]$ are expressed in Figure 3. Figure 4 shows rules $rc[l']$, $rc[u']$, and $rc[d']$. In the figure the new cells produced by $d'$ and $l'$ in rules $rc[d']$ and $rc[l']$, respectively, are connected to a new matrix object (orange object). As it is shown in Lemma 1, the next cell objects of $d'$ are $R$, $D$, $L$, or $E$ and $l'$ are $D$, $U$, or $L$, which are connected to the newly created matrix object at the north east of the original one as seen in Figure 5.

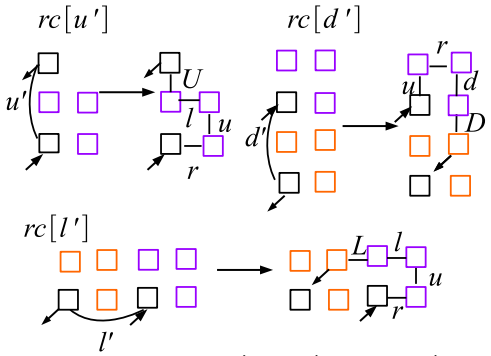



**Fig. 4.** Rules $rc[l']$, $rc[u']$, and $rc[d']$.

Figure 5 shows rules $rm[x]$ $x \in \{R, L, U, D\}$ and $rc[R']$. The uppercase cell objects $R, L, U$, and $D$ make 3 new matrix objects at east, north east, and north of the matrix object to which the start point of the cells connects. And the start point of the cells moves to the north east matrix object.

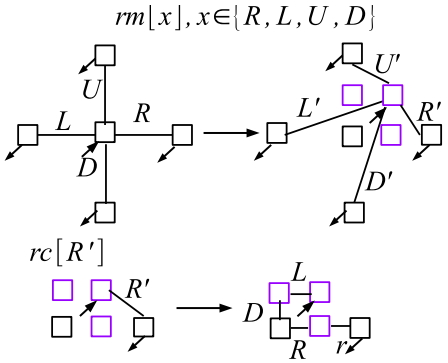



**Fig. 5.** Rules $rm[x]$ $x \in \{R, L, U, D\}$ and $rc[R']$.

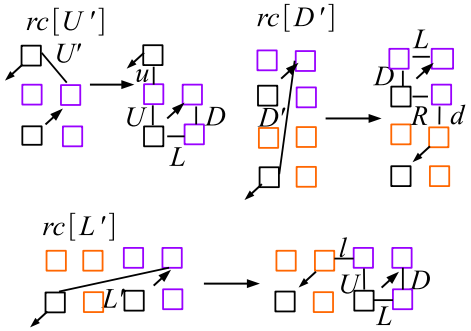



**Fig. 6.** Rules $rc[L']$, $rc[U']$, and $rc[D']$

Figure 6 shows rules $rc[L']$, $rc[U']$, and $rc[D']$. The uppercase and primed cell objects $R', L', U'$, and $D'$ divide into 4 cells. The new cells connect from the moved start point to the old end point.

Finally Figure 7 shows rules $rm[e]$, $rm[E]$, $rc[e']$, and $rc[E']$. The end cell objects, $e$ and $E$, make 3 new matrix objects at east, north east, and north of the object to which $e$ or $E$ connects. Then $e'$ or $E'$ divides into 4 cell objects and they connect the matrix objects. By Lemma 1, only $Re$ or $dE$ appears in the derivation.

**Lemma 1.** *For a cell object $x$ in $\Sigma$, the next cell object generated in the AP system is one of the next list:*

$$r\{u,d,r\}, \quad d\{R,L,D,E\}, \quad u\{r,l,u\}, \quad l\{D,U,L\}$$

$$R\{u,d,r,e\}, \quad D\{R,L,D\}, \quad U\{r,l,u\}, \quad L\{D,U,L\}.$$

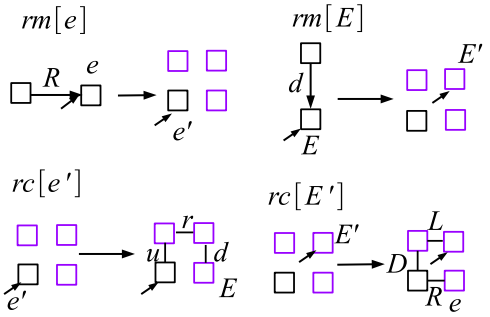



**Fig. 7.** In $rm[e]$ and $rm[E]$, the length of an edge in the produced square is half of the length of $R$ or $u$. Cell objects $R$ and $d$ are involved in the rules to give the length.

*Proof*. Since the two rules of cell objects $x$ and $x'$ simulate the rule of $x$ in the D0L system, we can verify the changes of the sequences of cell objects in the AP system by the rules in the D0L system.

In the first derivation of the D0L system, two symbols of $ur$, $rd$, and $dE$ are generated, which are all contained in the list of this lemma.

The sequences of two symbols insisted in this lemma are rewritten by the D0L system as follows:

$$ru \Rightarrow urdRrulU, \quad rd \Rightarrow urdRurdD, \quad rr \Rightarrow urdRurdR$$

$$dR \Rightarrow urdDLDRr, \quad dL \Rightarrow urdDDLUl, \quad dD \Rightarrow urdDLDRd \quad dE \Rightarrow urdDLDRe$$

$$\begin{array}{lll} ur \Rightarrow rulUurdR, & ul \Rightarrow rulUrulL, & uu \Rightarrow rulUrulU \\ lD \Rightarrow rulLLDRd, & lU \Rightarrow rulLDLUu, & lL \Rightarrow rulLDLUl \end{array}$$

$$Ru \Rightarrow LDRrrulU, Rd \Rightarrow LDRrurdD, Rr \Rightarrow LDRrurdR, Re \Rightarrow LDRrurdE$$

$$\begin{array}{lll} DR \Rightarrow LDRdLDRr, & DL \Rightarrow LDRdDLUl, & DD \Rightarrow LDRdLDRd \\ Ur \Rightarrow DLUuurdR, & Ul \Rightarrow DLUurulL, & Uu \Rightarrow DLUurulU \\ LD \Rightarrow DLUlLDRd, & LU \Rightarrow DLUlDLUu, & LL \Rightarrow DLUlDLUl \end{array}$$

All two symbols in the right-hand sides are contained in the list of two symbols of this lemma. Therefore, inductively the sequences of two cell objects are closed in the list of this lemma by any derivation in the system. □

By the rules in Figures 4 and 5, the next symbols of $d$ and $l$ must be capital symbols. Likewise, the next symbol of $D$ and $L$ must be capital symbols by Figures 5 and 6. Figure 7 requires that just before symbols of $e$ and $E$ are $R$ and $d$, respectively. Lemma 1 guarantees these requirements are satisfied. Thus we have the next theorem.

**Theorem 2.** *There are no gaps in the shapes assembled by the AP system, that is, the shape of the cell objects coincides with an approximated Hilbert curve in the chain code expression.*

## 5 Conclusion

We have introduced assembly P (AP) systems with rules producing matrix objects and absolute coordinates for the matrix objects. We have shown that there is an AP system which generates the sequence of the approximated Hilbert curves. The AP system places the matrix objects at the vertices of the approximated Hilbert curves and cells as the edges connecting the vertices. Thus the system defines the approximated Hilbert curves independent of any rendering systems. The system continues to generate the approximate Hilbert curves unlimitedly.

AP systems will be able to generate other space-filling curves. There will be AP systems which perform other computational tasks, for example, simulating a given Turing machine.

## References

1. Bader, M.: Space-filling Curves. Sprinver-Verlag, Berlin Heidelberg (2013)
2. Ceterchi, R., K.G.Subramanian: Generating pictures in string representation with P systems; the case of space-filling curves. Journal of Membrane Computing **2**, 369—379 (2020)
3. Gu, Z.: Construction, transformation and structures of $2 \times 2$ space-filling curves. arXiv:2412.16962v1 (2024)
4. Jin, G., Mellor-Curmmey, J.: SFCGen: A framework for efficient generation of multi-dimensional space-filling curves by recursion. ACM Transactions on Mathematical Software (TOMS) p. 120—148 (2005)
5. Nishida, T.: Assembly membrane systems. Journal of Membrane Computing **7**, 521–531 (2025)
6. Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer-Verlag, New York (1990)
7. Rozenberg, G., Salomaa, A.: The Mathematical Theory of L Systems. Academic Press, New York (1980)
8. Siromony, R., Subramanian, K.: Space-filling curves and infinite graphs. In: Graph-Grammars and Their Application to Computer Science, LNCS153. pp. 380–391 (1983)

# Appendix

## A Proof of Theorem 1

We prove Theorem 1 by showing that the strings generated by D0L system $G$ satisfy the recurrent equation (1) in Introduction. The formal definitions of the mappings $m$ and $\lambda$ are given by:

(i) Mirror $m$: $m(r) = l$, $m(l) = r$, $m(u) = u$, $m(d) = d$.
(ii) Rotate left $\lambda$: $\lambda(r) = u$, $\lambda(l) = d$, $\lambda(u) = l$, $\lambda(d) = r$.
(iii) Rotate right $(= \lambda^3)$: $\lambda^3(r) = d$, $\lambda^3(l) = u$, $\lambda^3(u) = r$, $\lambda^3(d) = l$.

The compositions $((\alpha \circ \beta)(x) = \beta(\alpha(x)))$ of the mappings are given by:

(i) $(m \circ \lambda)(r) = d$, $(m \circ \lambda)(l) = u$, $(m \circ \lambda)(u) = l$, $(m \circ \lambda)(d) = r$.
(ii) $(m \circ \lambda^3)(r) = u$, $(m \circ \lambda^3)(l) = d$, $(m \circ \lambda^3)(u) = r$, $(m \circ \lambda^3)(d) = l$.
(iii) $\lambda^2(r) = l$, $\lambda^2(l) = r$, $\lambda^2(u) = d$, $\lambda^2(d) = u$.

### A.1 Properties of the mappings over chain codes and recurrent equations over upper and lower chain codes

The next properties of the mappings are useful in the calculations over the recursive equation.

- $\lambda^4 = I$, $m^2 = I$.
- $m \circ \lambda \circ m = \lambda^3$ $(m \circ \lambda \circ m \circ \lambda = I)$.
- $m \circ \lambda^3 \circ m = \lambda$ $(m \circ \lambda^3 \circ m \circ \lambda = \lambda^2$, $m \circ \lambda^3 \circ m \circ \lambda^3 = I)$.
- $\lambda^2 \circ m \circ \lambda = m \circ \lambda^3$, $\lambda^2 \circ m \circ \lambda^3 = m \circ \lambda$.

Because we use the chain codes in upper and lower cases, we define the case change operator $\overline{x}$ as: $\overline{rlude} = RLUDE$ and $\overline{RLUDE} = rlude$.

We rewrite the recurrent equation (1) into equations over chain codes of upper and lower cases. For the sake of simplicity, we use the symbol $H_n$ to express the approximate strings of the Hilbert curves with chain code of upper and lower case symbols. For even $n$

$$H_n = (m \circ \lambda^3)(H_{n-1})UH_{n-1}RH_{n-1}D(m \circ \lambda)(\overline{H_{n-1}}).$$

For odd $n$

$$H_n = (m \circ \lambda^3)(H_{n-1})uH_{n-1}rH_{n-1}d(m \circ \lambda)(\overline{H_{n-1}}).$$

It should be noted that the strings defined by the above recurrent equations have the same graphical meanings as the strings defined by recurrent equation (1).

By direct calculations, the two step recurrent equations on $H_n$ are obtained. For even $n$,

$$\begin{aligned}
H_n &= (m \circ \lambda^3)(H_{n-1})U\, H_{n-1}R\, H_{n-1}D\, (m \circ \lambda)(\overline{H_{n-1}}) \\
&= (m \circ \lambda^3)((m \circ \lambda^3)(H_{n-2})u\, H_{n-2}r\, H_{n-2}d\, (m \circ \lambda)(\overline{H_{n-2}}))U \\
&\quad (m \circ \lambda^3)(H_{n-2})uH_{\,n-2}rH_{\,n-2}d\, (m \circ \lambda)(\overline{H_{n-2}})R \\
&\quad (m \circ \lambda^3)(H_{n-2})uH_{\,n-2}rH_{\,n-2}d\, (m \circ \lambda)(\overline{H_{n-2}})D \\
&\quad (m \circ \lambda)(\overline{(m \circ \lambda^3)(H_{n-2})uH_{\,n-2}r\, H_{n-2}d\, (m \circ \lambda)(\overline{H_{n-2}})}) \\
&= H_{n-2}r\, (m \circ \lambda^3)(H_{n-2})u\, (m \circ \lambda^3)(H_{n-2})l\, \lambda^2(\overline{H_{n-2}})U \\
&\quad (m \circ \lambda^3)(H_{n-2})u\, H_{n-2}r\, H_{n-2}d\, (m \circ \lambda)(\overline{H_{n-2}})R \\
&\quad (m \circ \lambda^3)(H_{n-2})u\, H_{n-2}r\, H_{n-2}d\, (m \circ \lambda)(\overline{H_{n-2}})D \\
&\quad \lambda^2(\overline{H_{n-2}})L\, (m \circ \lambda)(\overline{H_{n-2}})D\, (m \circ \lambda)(\overline{H_{n-2}})R\, H_{n-2}.
\end{aligned}$$

For odd $n$

$$\begin{aligned}
H_n &= (m\circ\lambda^3)H_{n-1}u\,H_{n-1}r\,H_{n-1}d\,(m\circ\lambda)(\overline{H_{n-1}})\\
&= (m\circ\lambda^3)((m\circ\lambda^3)(H_{n-2})U\,H_{n-2}R\,H_{n-2}D\,(m\circ\lambda)(\overline{H_{n-2}}))u\\
&\quad (m\circ\lambda^3)(H_{n-2})UH_{\,n-2}RH_{\,n-2}D\,(m\circ\lambda)(\overline{H_{n-2}})r\\
&\quad (m\circ\lambda^3)(H_{n-2})UH_{\,n-2}RH_{\,n-2}D\,(m\circ\lambda)(\overline{H_{n-2}})d\\
&\quad (m\circ\lambda)(\overline{(m\circ\lambda^3)(H_{n-2})UH_{\,n-2}R\,H_{n-2}D\,(m\circ\lambda)(\overline{H_{n-2}})})\\
&= H_{n-2}R\,(m\circ\lambda^3)(H_{n-2})U\,(m\circ\lambda^3)(H_{n-2})L\,\lambda^2(\overline{H_{n-2}})u\\
&\quad (m\circ\lambda^3)(H_{n-2})U\,H_{n-2}R\,H_{n-2}D\,(m\circ\lambda)(\overline{H_{n-2}})r\\
&\quad (m\circ\lambda^3)(H_{n-2})U\,H_{n-2}R\,H_{n-2}D\,(m\circ\lambda)(\overline{H_{n-2}})d\\
&\quad \lambda^2(\overline{H_{n-2}})l\,(m\circ\lambda)(\overline{H_{n-2}})d\,(m\circ\lambda)(\overline{H_{n-2}})r\,H_{n-2}.
\end{aligned}$$

Before proceeding the proof, we introduce a few notations to make expressions concise. For sequences of strings $s_1,\dots,s_k$ and $t_1,\dots,t_k$, let

$$(s_1,\dots,s_k)\uplus(t_1,\dots,t_k) = s_1t_1s_2t_2\cdots s_kt_k.$$

For mappings $y_1,\dots,y_k$ and a string $H_n$, let

$$(y_1,\dots,y_k)(H_n) = (y_1(H_n),\dots,y_k(H_n)).$$

To write a long vector concisely, we write a vector $(x_1,\dots,x_k)$ as

$$\begin{pmatrix} x_1, & \dots, & x_{l_1},\swarrow \\ x_{l_1+1}, & \dots, & x_{l_2},\swarrow \\ \vdots & \ddots & \vdots \\ x_{l_p+1}, & \dots, & x_k \end{pmatrix}.$$

Using the above notations, the two step recurrent equation for even $n$ becomes

$$H_n = Y(H_{n-2})\uplus A$$

where

$$Y = \begin{pmatrix} I, & (m\circ\lambda^3), & (m\circ\lambda^3), & \lambda^{2-},\swarrow \\ (m\circ\lambda^3), & I, & I, & (m\circ\lambda)^-,\swarrow \\ (m\circ\lambda^3), & I, & I, & (m\circ\lambda)^-,\swarrow \\ \lambda^{2-}, & (m\circ\lambda)^-, & (m\circ\lambda)^-, & I \end{pmatrix},$$

$$A = \begin{pmatrix} r,\ u,\ l,\ U,\swarrow \\ u,\ r,\ d,\ R,\swarrow \\ u,\ r,\ d,\ D,\swarrow \\ L,\ D,\ R, \quad \epsilon \end{pmatrix},$$

and $y^-(H) = y(\overline{H})$. And for odd $n$, we have

$$H_n = Y(H_{n-2})\uplus B$$

with

$$B = \begin{pmatrix} R,\ U,\ L,\ u,\swarrow \\ U,\ R,\ D,\ r,\swarrow \\ U,\ R,\ D,\ d,\swarrow \\ l,\ d,\ r, \quad \epsilon \end{pmatrix} = \overline{A}.$$

### A.2 The relations between recurrent equations and $\tau$

Next we consider the relation between the two step recurrent equations and the morphism $\tau$ in $G$. Using the upper-lower exchange on the chain codes, $\tau$ can be expressed by:

$$\begin{aligned} x &\to urd\overline{x} \quad \text{for } x \in \{r, d, e\} \\ y &\to rul\overline{y} \quad \text{for } y \in \{l, u\} \\ \overline{x} &\to LDRx \quad \text{for } x \in \{r, d, e\} \\ \overline{y} &\to DLUy \quad \text{for } y \in \{l, u\}. \end{aligned}$$

Now the next lemmas hold.

**Lemma 2.** *For even $n$,*

*(1)* $\tau^2(H_n x_1) = H_{n+2}x_1$, $x_1 \in \{r, d, e\}$.
*(2)* $\tau^2((m \circ \lambda^3)(H_n)x_2) = (m \circ \lambda^3)(H_{n+2})x_2$, $x_2 \in \{l, u\}$.
*(3)* $\tau^2(\lambda^2(\overline{H_n})x_3) = \lambda^2(\overline{H_{n+2}})x_3$, $x_3 \in \{L, U\}$.
*(4)* $\tau^2((m \circ \lambda)(\overline{H_n})x_4) = (m \circ \lambda)(\overline{H_{n+2}})x_4$, $x_4 \in \{R, D, E\}$.

*Proof*. We prove this lemma by induction on $n$.
The base is the case $n = 0$ and is proved by the next equations. Please note that $H_0 = \epsilon$.

(1) For $x_1 \in \{r, d, e\}$, $\tau^2(H_0x_1) = \tau(\tau(x_1)) = \tau(urd\overline{x_1}) =$
$rulU\, urdR\, urdD\, LDRx_1 = H_2x_1$.
(2) For $x_2 \in \{l, u\}$, $\tau^2((m \circ \lambda^3)(H_0)x_2) = \tau(\tau(x_2)) = \tau(rul\overline{x_2}) =$
$urdR\, rulU\, rulL\, DLUx_2 = (m \circ \lambda^3)(H_2)x_2$.
(3) For $x_3 \in \{L, U\}$, $\tau^2(\lambda^2(\overline{H_0})x_3) = \tau(\tau(x_3)) = \tau(DLU\overline{x_3}) =$
$LDRd\, DLUl\, DLUu\, rulx_3 = \lambda^2(\overline{H_2})x_3$.
(4) For $x_4 \in \{R, D, E\}$, $\tau^2((m \circ \lambda)(\overline{H_0})x_4) = \tau(\tau(x_4)) = \tau(LDR\overline{x_4}) =$
$DLUl\, LDRd\, LDRr\, urdx_4 = (m \circ \lambda)(\overline{H_2})x_4$.

Next let us assume equations (1) — (4) hold for some $n - 2$ with $n \geq 2$.

For $x_1 \in \{r, d, e\}$, we have the next equation by the two step recurrent equation

$$\tau^2(H_n x_1) = \tau^2((Y(H_{n-2}) \uplus A)x_1).$$

Since all 16 substrings of the form $\xi(H_{n-2})\alpha$ with $\xi \in \{I, m \circ \lambda^3, (m \circ \lambda)^-, \lambda^{2-}\}$ and $\alpha \in \Sigma$ are contained in the inductive hypotheses, we have

$$\tau^2(H_n x_1) = (Y(H_n) \uplus A)x_1.$$

Then the two step recurrent equation implies

$$\tau^2(H_n x_1) = H_{n+2}x_1.$$

For $x_2 \in \{l, u\}$, we have

$$\begin{aligned} \tau^2((m \circ \lambda^3)(H_n)x_2) &= \tau^2((m \circ \lambda^3)(Y(H_{n-2}) \uplus A)x_2) \\ &= \tau^2(((Y \circ m \circ \lambda^3)(H_{n-2}) \uplus (m \circ \lambda^3)(A))x_2). \end{aligned}$$

By direct calculations, we have

$$Y \circ m \circ \lambda^3 = \begin{pmatrix} m\circ\lambda^3, & I, & I, & (m\circ\lambda)^- \swarrow \\ I, & m\circ\lambda^3, & m\circ\lambda^3, & \lambda^{2-} \swarrow \\ I, & m\circ\lambda^3, & m\circ\lambda^3, & \lambda^{2-} \swarrow \\ (m\circ\lambda)^-, & \lambda^{2-}, & \lambda^{2-}, & m\circ\lambda^3 \end{pmatrix} = Y_2$$

and

$$(m\circ\lambda^3)(A) = \begin{pmatrix} u, & r, & d, & R \swarrow \\ r, & u, & l, & U \swarrow \\ r, & u, & l, & L \swarrow \\ D, & L, & U, & \epsilon \end{pmatrix} = A_2.$$

Using the equations, we have

$$\tau^2((m\circ\lambda^3)(H_n)x_2) = \tau^2((Y_2(H_{n-1}) \uplus A_2)x_2).$$

The inductive hypotheses are applicable to all 16 substrings in $(Y_2(H_{n-2}) \uplus A_2)x_2$, we get

$$\begin{aligned} \tau^2((m\circ\lambda^3)H_n x_2) &= (Y_2(H_n) \uplus A_2)x_2 \\ &= (m\circ\lambda^3)(Y(H_n) \uplus A)x_2 \\ &= (m\circ\lambda^3)(H_{n+2})x_2. \end{aligned}$$

The last equation is due to the two step recurrent equation.

Likewise, for $x_3 \in \{L, U\}$,

$$\begin{aligned} \tau^2(\lambda^{2-}(H_n)x_3) &= \tau^2(\lambda^{2-}(Y(H_{n-2} \uplus A)x_3)) \\ &= \tau^2((Y_3(H_{n-2}) \uplus A_3)x_3) \\ &= (Y_3(H_n) \uplus A_3)x_3 \\ &= \lambda^{2-}(Y(H_n) \uplus A)x_3 \\ &= \lambda^{2-}(H_{n+2})x_3, \end{aligned}$$

and for $x_4 \in \{R, D, E\}$,

$$\begin{aligned} \tau^2((m\circ\lambda)^-(H_n)x_4) &= \tau^2((m\circ\lambda)^-(Y(H_{n-1}) \uplus A)x_4) \\ &= \tau^2((Y_4(H_{n-2}) \uplus A_4)x_4) \\ &= (Y_4(H_n) \uplus A_4)x_4 \\ &= (m\circ\lambda)^-(Y(H_n) \uplus A)x_4 \\ &= (m\circ\lambda)^-(H_{n+2})x_4, \end{aligned}$$

with

$$\begin{aligned} Y_3 &= Y \circ \lambda^{2-} \\ &= \begin{pmatrix} \lambda^{2-}, & (m\circ\lambda)^-, & (m\circ\lambda)^-, & I \swarrow \\ (m\circ\lambda)^-, & \lambda^{2-}, & \lambda^{2-}, & m\circ\lambda^3, \swarrow \\ (m\circ\lambda)^-, & \lambda^{2-}, & \lambda^{2-}, & m\circ\lambda^3, \swarrow \\ I, & m\circ\lambda, & m\circ\lambda^3, & \lambda^{2-} \end{pmatrix} \end{aligned}$$

$$A_3 = \lambda^{2-}(A)$$
$$= \begin{pmatrix} L,\ D,\ R,\ d, \swarrow \\ D,\ L,\ U,\ l, \swarrow \\ D,\ L,\ U,\ u, \swarrow \\ r,\ u,\ l,\ \epsilon \end{pmatrix}$$

and

$$Y_4 = Y \circ (m \circ \lambda^3)^-$$
$$= \begin{pmatrix} (m \circ \lambda)^-, & \lambda^{2-}, & \lambda^{2-}, & m \circ \lambda, \swarrow \\ \lambda^{2-}, & (m \circ \lambda)^-, & (m \circ \lambda)^-, & I, \swarrow \\ \lambda^{2-}, & (m \circ \lambda)^-, & (m \circ \lambda)^-, & I, \swarrow \\ m \circ \lambda^3, & I, & I, & (m \circ \lambda)^- \end{pmatrix}$$
$$A_4 = (m \circ \lambda)^-(A)$$
$$= \begin{pmatrix} D,\ L,\ U,\ l, \swarrow \\ L,\ D,\ R,\ d, \swarrow \\ L,\ D,\ R,\ r, \swarrow \\ u,\ r,\ d,\ \epsilon \end{pmatrix}.$$

Thus equations (1) — (4) hold for $n$. □

**Lemma 3.** *For every odd $n$, $\tau^2(H_n E) = H_{n+2}E$.*

*Proof* By (one step) recurrent equation,

$$\tau^2(H_n E) = \tau^2((m \circ \lambda^3)(H_{n-1})u\, H_{n-1} r\, H_{n-1} d\, (m \circ \lambda)(\overline{H_{n-1}})E).$$

Because $n-1$ is even, Lemma 1 implies

$$\tau^2(H_n E) = (m \circ \lambda^3)(H_{n+1})u\, H_{n+1} r\, H_{n+1} d\, (m \circ \lambda)(\overline{H_{n+1}})E = H_{n+2}E.$$

The second equation is due to the one step recurrent equation. □

Then Theorem 1 directly follows from Lemmas 2 and 3.

# Spiking Neural P systems with colored spikes and multiple channels

José M. Sempere[1,2] and Rodrigo Llanes[1]

[1] Valencian Research Institute for Artificial Intelligence (VRAIN)
[2] Valencian Graduate School and Research Network of Artificial Intelligence (VALGRAI)
Universitat Politècnica de València, Spain
jsempere@dsic.upv.es

**Abstract.** In this paper, we combine two variants of Spiking Neural (SN) P systems: we use a non-singleton alphabet of spikes (colored spikes), and we use multiple channels that can be used in the rules of the neurons. The completeness of the proposed model is shown by the simulation of register machines. We also show the equivalence between the proposed model and other different ones, such as virus machines and neural-like P systems with plasmids. We show the simulations of the latter models by our proposed one. Hence, we can consider that our proposed model is a unifying one that can achieve two main properties: different encodings for the information (spikes), and different communication topologies (channels).

**Keywords:** SN P systems, colored spikes, multiple channels, completeness, virus machines, neural-like P systems with plasmids

## 1 Introduction

Membrane computing is a research area founded by Gheorghe Păun at the end of the last century [24], and it is a significant part of the natural computing paradigm [11]. The source of inspiration for membrane computing is the eukaryotic living cell whose internal structure allows regions separated by membranes, where different molecular reactions are carried out by the biomolecular machinery of the cell. Membrane computing models are known as P systems. Initially, P systems were proposed as cell-like transition P systems [23]. Then, tissue P systems were proposed through the association and abstraction of cell-like P systems [15]. A very few years later, the spiking neural (SN) P system was proposed [10] that allows rate and pulse encoding [8]. It was proposed by introducing artificial spiking neural networks in the framework of membrane computing.

Interest on SN P systems has increased in the last years, both when it comes to proposing new variants or new features to the basic model [12, 2], and when it comes to applying it to solve problems in many different areas [6] . Among the proposals that have been made during the last years, we could mention fuzzy reasoning spiking neural P systems (FRSNP) [28], Optimization Spiking neural

P Systems (OSNPS) [39], Spiking neural P systems with structural plasticity [3], electrical synaptic transmission-based spiking neural P systems (ESTSNPS) [36], Spiking Neural P Systems with budding rules [9], Spiking neural P systems with neuron division and budding [19], time-free Spiking Neural P Systems [21], and Spiking Neural P Systems with Astrocytes [20]. Two recent references on spiking neural P systems, and its variants and applications are [26, 40].

In this work, we use SN P systems that allows the sending/receiving of different spikes through different channels. The use of different channels in the rules is not novel: it was first proposed by Alhazov *et al.* [1], and referred again in [38]. It has been proposed again in [27], where the authors use different spiking channels to connect the neurons in the network. An increasing interest on the use of multiple channels has been shown in the last years [14]. So, for example, Miao *et al.* proposed the use of multiple channels with dynamic adjustment of the rules [16], while in [18], Ning *et al.* propose a combination of multiple channels with autapses [35].

The use of a non-singleton alphabet to define the spikes of the system (colored spikes) is not novel given that it was proposed in [34], and it has been recently used to solve NP-complete problems [22].

In this work, we combine the two previously referred features: multiple channels and colored spikes. We integrate them in the rules of the SN P system. So, the rules can manage different spikes at every moment, and the spikes can be sent to other neurons through different channels. These characteristics, when combined appropriately, provide us with an efficient way to simulate other models proposed in the field of membrane computing, such as virus machines [5], and neural-like P systems with plasmids [4]. As far as we are aware, the use of SN P systems to simulate these models is non-existent or has never even been considered. Furthermore, we believe that our proposal could become a general computing framework in which several networks operate simultaneously, influencing one another, with nodes (neurons) specialising in different types of information.

The structure of this work is as follows: In section 2, we provide the basic concepts and definition for membrane computing and spiking neural P systems. In section 3, we propose our model that combines a non-singleton alphabet for spikes, and multiple channels in the rules. We establish the completeness of the proposed model by simulating register machines. In section 4, we propose efficient simulations of other computational models by our model. We propose a simulation for virus machines, and a simulation for bacterial computing with plasmids. Finally, in section 5, we outline some conclusions of our work, and we point out to future research works.

## 2 Basic concepts and definitions

In the following, we assume that the reader is familiar with the basic concepts of formal language theory [32]. Nevertheless, we will introduce additional concepts on multisets, strings and languages.

Let $\mathbb{N}$ denotes the set of natural numbers. Given a set $V$, a *multiset* $M$ over $V$ is defined as a mapping $M : V \to \mathbb{N}$, where for every $a \in V$, $M(a)$ denotes the multiplicity of the element $a$ in $M$. Given two multisets over $V$, $M_1$ and $M_2$, we will say that $M_1 \subseteq M_2$ iff $(\forall a \in V)\ M_1(a) \leq M_2(a)$. Given an alphabet $\Sigma$, and the string $x \in \Sigma^*$, we denote the number of symbols $a \in \Sigma$ in $x$ by $|x|_a$. Given the alphabet $\Sigma = \{a_1, a_2, \ldots, a_n\}$, the Parikh projection of a string $x \in \Sigma^*$, is the vector $\Psi_\Sigma(x) = (|x|_{a_1}, |x|_{a_2}, \ldots, |x|_{a_n})$. Given any language $L \subseteq \Sigma^*$, the Parikh projection of L in $\Sigma$ is the set $\Psi_\Sigma(L) = \{\Psi_\Sigma(x) : x \in L\}$. For any set $V = \{a_1, a_2, \ldots, a_n\}$, any multiset $M$ over $V$ can be represented by the string $x = {a_1}^{M(a_1)} {a_2}^{M(a_2)} \ldots {a_n}^{M(a_n)}$ (observe that any permutation in the symbols of $x$ represents the multiset $M$).

Now, we introduce the definition of a register machine, that is a universal model of computation proposed in [17].

**Definition 1.** A register machine is defined by $M = (m, H, l_0, l_h, I)$, where:

1. $m \geq 1$ indicates the number of machine registers. Every register holds a positive integer value, and all the registers are initialized to zero.
2. $H$ is a set of instruction labels.
3. $l_0$ is the label corresponding to the initial instruction.
4. $l_h$ is the label corresponding to the halt instruction.
5. $I$ is the set of instructions such that $|I| = |H|$ and each label from $H$ refers to an instruction in $I$. Each of the instructions of $I$ is in one of the following forms:
   - $l_i : (\text{ADD}(r), l_j, l_k)$ add one to the register $r$, and then non-deterministically jumps to instruction $l_j$ or $l_k$.
   - $l_i : (\text{SUB}(r), l_j, l_k)$ subtract one if the contents of register $r$ is greater than zero, and jump to $l_j$. Otherwise, jump to $l_k$.
   - HALT finishes the computation.

□

The register machine begins its execution by the instruction labeled $l_0$, then it executes the instruction and it jumps to the next one according to the type of instruction and the value of the registers. This process is repeated until it reaches the halt instruction $l_h$. There is a predefined register where the result of the computation is stored when the machine halts. Usually, the first register is taken as the output register.

Basic concepts and results about P systems and membrane computing can be consulted at [25], while basic definitions related to Spiking Neural P systems can be taken from [10].

**Definition 2.** A spiking neural P system (SN P system, for short) of degree $m \geq 1$ is defined by $\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out)$, where:

1. $O = \{a\}$ is a singleton alphabet of spikes
2. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$, where
   (a) $n_i \geq 0$ is the initial number of spikes in $\sigma_i$.

(b) $R_i$ is a finite set of rules in one of the following two forms:
- firing or spiking rules: $E/a^c \rightarrow a;d$ where $E$ is a regular expression over $O$, and $c \geq 1,\ d \geq 0$ are integer numbers. We omit $E$ whenever it be equal to $a^c$, and we omit $d$ whenever it be equal to 0.
- forgetting rules: $a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each spiking rule $E/a^c \rightarrow a;d$ then $a^s \notin L(E)$ ($L(E)$ is the regular language defined by $E$)

3. $syn \subseteq \{1,2,\ldots m\} \times \{1,2,\ldots,m\}$ with $(i,i) \notin syn$ for $1 \leq i \leq m$, is the directed graph of synapses between neurons
4. $in,\ out\ \in \{1,\ldots,m\}$ indicate the input and the output neurons of $\Pi$.

□

Observe that we can omit the definition of the input neuron in order to define the SN P system as a generator model.

At neuron $\sigma_i$, the firing rules $E/a^c \rightarrow a;d$ are applied as follows: if the neuron contains $k \geq c$ spikes and $a^k \in L_E$ (the language denoted by the expression $E$) then $c$ spikes are removed from $\sigma_i$ and one spike is delivered to all the neurons $\sigma_j$ connected to $\sigma_i$ with $(i,j) \in syn$. If $d = 0$ the spike is immediately emitted, otherwise it is emitted after $d$ computation steps (during these computation steps, the neuron is closed, so it cannot receive spikes, it cannot apply the rules and, subsequently, it cannot send new spikes). At neuron $\sigma_i$, the forgetting rule $a^s \rightarrow \lambda$ is applied as follows: if the neuron $\sigma_i$ contains exactly $s$ spikes and no firing rule can be applied then all the spikes of the neuron are removed.

A configuration of the system at time $t$ during a computation is defined by the tuple $(i_1/t_1,\ldots,i_m/t_m)$ that denotes the number of spikes that are at every neuron together with the computation time needed to open the neuron. The initial configuration of the SN P system is $(n_1/0,\ldots,n_m/0)$. During a computation, the step when a spike is emitted by the output neuron can be marked by '1' while the other steps are marked by '0'. The binary sequence that is obtained in such a way during the computation is called the spike train of the system. It should be noted that there are multiple ways to read the system output. For example, in generator systems, if the computation does not finish, the output is the number of spikes in the neuron *out* at each computation step. The total number of spikes that the system has sent out to the environment during a computation (which must be finite) can also be considered as output.

**Example 1.** In the figure 1 we show an example of a SN P system with three neurons. The output neuron is neuron 3. The input neuron is neuron 1. Neuron 1 has $n$ initial spikes and one firing rule. Neurons 2 and 3 have no initial spikes and one firing rule.

In the first iteration, only the rule $a^+/a \rightarrow a$ of neuron 1 can be applied. It consumes one spike and sends a spike to neurons 2 and 3. In the following $n-1$ iterations, the three neurons will be active. Observe that along the time, neuron 3 receives $2 \times n$ spikes that will be sent to the environment. Once all these spikes have been consumed the system halts and it has generated $2 \times n$ spikes.

□

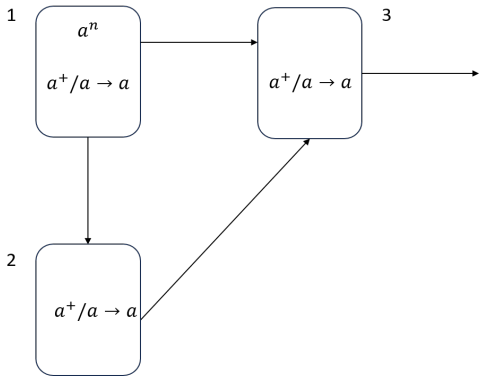



**Fig. 1.** An example of an SN P system that generates $2 \times n$.

Once we have defined the SN P system, we introduce two variants of it: multiple channels [27], and a spike alphabet with more than one symbol (colored spikes) [34].

**Definition 3.** [27] A spiking neural P system with multiple channels (SNP-MC system, for short) of degree $m \geq 1$ is defined by $\Pi = (O, L, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, out)$, where:

1. $O = \{a\}$ is a singleton alphabet of spikes
2. $L = \{1, 2, \ldots, N\}$ is the alphabet of channel labels
3. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, L_i, R_i)$, $1 \leq i \leq m$, where:
   (a) $n_i \geq 0$ is the initial number of spikes contained in $\sigma_i$.
   (b) $L_i \subseteq L$ is a finite set of channel labels used in the neuron
   (c) $R_i$ is a finite set of extended rules in the form $E/a^c \to a^p(l)$ where $E$ is a regular expression over $O$, and $c \geq 1,\ p \geq 0, l \in L_i$.
4. $syn = \{(i, j, l)\} \subseteq \{1, 2, \ldots m\} \times \{1, 2, \ldots, m\} \times L$ with $(i, i, l) \notin syn$ for $1 \leq i \leq m$, and $l \in L$ (synapse connections);
5. $out \in \{1, \ldots, m\}$ is the output neuron.

□

Observe that the rules of the system are defined in a similar way as they have been described in the definition 1. The forgetting rules can be established as those extended rules with $p = 0$ and no channel reference. Every rule establishes the channels that it uses to send the spikes. The spikes are only sent to the connected neurons by the channel established in the rule. In addition, in each computation step there can be some competition between different rules at the neuron. Provided that more than one rule can be applied, then the system selects only one of them non-deterministically. So, the neurons work in sequential manner, that is they apply only one rule at every computation step.

It has been proved that SNP-MC systems are universal, even if they work in a sequential manner [14].

**Definition 4.** [34] A spiking neural P system with colored spikes (SNP-CS system, for short) of degree $m \geq 1$ is defined by $\Pi = (C, O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out)$, where:

1. $C = \{1, 2, \ldots g\}$ is a finite set of colors to mark the color of a spike. Every spike is associated with a unique color
2. $O = \{a_1, a_2, \ldots, a_g\}$ is an alphabet of $g$ colored spikes
3. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (< n_1^i, n_2^i, \ldots, n_g^i >, R_i)$, $1 \leq i \leq m$, where:
   (a) $n_h^i \in \mathbb{N}$ is the number of spikes $a_h \in O$ initially placed in neuron $\sigma_i$.
   (b) $R_i$ is a finite set of spiking and forgetting rules in the form:
      – Spiking rule: $E/a_1^{c_1} a_2^{c_2} \ldots a_g^{c_g} \rightarrow a_1^{p_1} a_2^{p_2} \ldots a_g^{c_g}; d$ where $E$ is a regular expression over $O$, and $c_i, p_i \geq 0$;
      – Forgetting rule: $a_1^{s_1} a_2^{s_2} \ldots a_g^{s_g} \rightarrow \lambda$ where $a_1^{s_1} a_2^{s_2} \ldots a_g^{s_g} \notin L(E)$ for any regular expression $E$ associated with a spiking rule;
4. $syn \subseteq \{1, 2, \ldots m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$;
5. $in, out \in \{1, \ldots m\}$ are the input and the output neurons

□

Observe that the SNP-CS systems are a generalization of the SN P systems. The functioning of the rules in the neurons, the behavior of the SNP-CS system and its output are generalizations of the SN P systems. Hence, the model is computationally complete. Note that, in this case, the membership problem associated with any regular expressions $E$ in the spiking rules should be replaced by a membersip problem within the Parikh projection $\Psi_O(L_E)$.

## 3 Spiking neural P systems with multiple channels and colored spikes

We introduce a new model of SN P systems that combines the use of multiple channels, together with a non-singleton alphabet of spikes. We define different ways of obtaining the system outputs, and we will prove the completeness of the model by proposing a simulation of any register machine program.

**Definition 5.** A spiking neural P system with multiple channels and colored spikes (SNP-MC-CS system, for short) of degree $m \geq 1$ is defined by $\Pi = (O, L, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in)$, where:

1. $O = \{a_1, a_2, \ldots, a_g\}$ is an alphabet with $g$ colored spikes
2. $L = \{1, 2, \ldots, k\}$ is an alphabet of channel labels
3. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (w_i, R_i)$, $1 \leq i \leq m$, where:
   (a) $w_i \in O^*$ is a string that denotes the initial multiset of spikes in the neuron.
   (b) $R_i$ is a finite set of rules in the forms:
      – Firing rules: $E/w_c \rightarrow w_1\,(l_1)\, w_2\,(l_2) \ldots w_n\,(l_n)\,; d$ where $E$ is a regular expression over $O$, and $w_c, w_i \in O^*$ $1 \leq i \leq n, l_i \in L$, $d \geq 0$.

- Forgetting rules: $w_c \to \lambda$, where $w_c \in O^*$.

4. $syn \subseteq \{1, 2, \ldots m\} \times \{1, 2, \ldots, m, out\} \times L$ is the synapse connections with channels where $(i, i, l) \notin syn$ for $1 \leq i \leq m$ and $\forall\ l\ \in L$. Observe that $(i, out, l)$ denotes that the neuron $\sigma_i$ sends the spikes out to the environment by the channel $l$.
5. $in\ \in \{1, \ldots, m\}$ is the input neuron. Observe that the input neuron can be omitted whenever the system is a generator.

□

The firing rules $E/w_c \to w_1\,(l_1)\,w_2\,(l_2) \ldots w_n\,(l_n)\,;d$ of the neuron $\sigma_i$ can be applied whenever $\psi_O(x_i) \in \Psi_O(L_E)$ and $w_c \subseteq x_i$, where $x_i$ denotes the spikes in the neuron $\sigma_i$ and $L_E$ is the language denoted by the regular expression $E$. In such a case, the spikes denoted by $w_c$ are removed from the neuron, and the spikes denoted by $w_i$ are sent to the connected neurons (or to the environment) by the channel $l_i$ according to the synaptic connections of the neuron. If the delay $d = t > 0$, then the neuron is blocked, and it can neither send nor receive spikes after $t$ computation steps.

The forgetting rules of the neuron $\sigma_i$ in the form $w_c \to \lambda$, can be applied whenever no firing rule is applicable and $w_c \subseteq x_i$, where $x_i$ denotes the spikes in the neuron $\sigma_i$. In such a case, the spikes denoted by $w_c$ are removed from the neuron.

The system runs in a maximally parallel manner, that is, at each computational step, every neuron that has some applicable rule applies it. A rule is applicable iff the spikes in the neuron meet the regular expression in the rule, and the number of spikes are greater or equal to the number of spikes to be removed. In the case of forgetting rules, the regular expression condition is not considered. If there are more than one applicable rule, the firing rules will have priority over those of forgetting. If there are more than one applicable firing rule in conflict, it means that they require common spikes, then, only one of them is non-deterministically selected to be applied. If a rule with a delay is applied, the neuron will be blocked just as in SN P systems. If several rules with delays are applied at the same computational step, the neuron will be blocked for the maximum delay value of the applied rules. Finally, it should be noted that the computation finishes whenever there is no blocked neuron, and no rule can be applied in any neuron.

Regarding the output of the system, we will propose four different ways to obtain it, as follows:

- *halting mode*: The result of the computation is the number of spikes that have been sent to the environment during the computation, no matter their colors. The output is defined only when the system halts.
- *multi-channel halting mode*: As in the halting mode, the output of the model is the set of spikes sent to the environment during the computation. The output spikes are defined for the channels through which they have been sent. In this case, the output is a vector of natural numbers.

- *temporary mode*: This mode is the one used for generating systems, since it is not necessary reaching a halting configuration, the system output will be read as the sequence of spikes sent to the environment at every computation step (a spike train).
- *multi-channel temporary mode*: It can be used for signal processing, as it allows reading the output as the sequence of spikes (a spike train) sent to the environment at every computation step for each of the channels.

**Example 2.** In the figure 2, we show an example of an SNP-MC-CS system with two neurons, 1 (left) and 2 (right). The system uses three channels, and it does not use any input neuron, so it is a generative model. In the first computation step, the system can only execute the rule in the neuron 2, sending a spike $a$ to the neuron 1 through channel (1). In the second computation step, the following rules can be chosen non-deterministically: $a \to a(0), ab(2)$, so spikes $a$ and $b$ will be sent to the environment through the channel (2), and a spike $a$ is sent to the neuron 2 through the channel (0). So in the next computation step the system will repeat the initial configuration, the second rule, $a \to ab(2)$ sends the spikes $a$ and $b$ to the environment and the system halts, given that no rule can be applied again. The system outputs in a halting mode the multiset defined by $\Psi_{\{a,b\}}(\{a^n b^n : n \geq 1\})$.

□

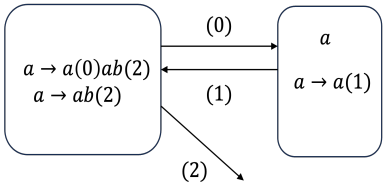



**Fig. 2.** An example of an SNP-MC-CS system.

### 3.1 Completeness

In the following, we show that the SNP-MC-CS systems are complete models of computation. It is important to highlight that SNP-CS systems are universal [34], and they are a particular case of SNP-MC-CS with a single channel. On the other hand, SNP-MC are also universal [27], and they are a particular case of SNP-MC-CS with a single color. So, the completeness of the SNP-MC-CS system is trivially deduced from the referred previous works, as established in the following theorem.

**Theorem 1.** SNP-MC-CS systems are universal models of computation.

*Proof.* Trivially deduced from [27] and [34]. □

Nevertheless, we propose the simulations of the basic instructions of register machines by SNP-MC-CS systems. In the following, we associate a neuron with label $l_i$ with each instruction with a label $l_i$, and a neuron with label $r$ with the register $r$ in the machine. If the contents of any register is a natural number $p$, the multiset $c^p$ will be stored in its associated neuron. The activation of any instruction will be triggered by the spike $a$. In addition, other spikes are used to simulate the SUB instructions in the registers. The HALT instruction will be managed by using an additional spike $h$.

Now, we will define a scheme to implement every basic instruction of the register machine as a SNP-MC-CS sub-system.

– Simulation of the instruction $l_i : (\text{ADD}(r), l_j, l_k)$
The ADD instruction can be simulated by the system proposed in the figure 3. The simulation works as follows: upon receiving a spike $a$ in neuron $l_i$ (which represents the instruction with the same label), one of the two rules is executed non-deterministically. Observe that in both rules a spike $c$ is sent to the neuron $r$ through the channel (1). It adds one unit to the contents of the register $r$. A spike $a$ is sent to the neurons to $l_j$ or $l_k$ in order to activate the following instruction in a non-deterministic manner.

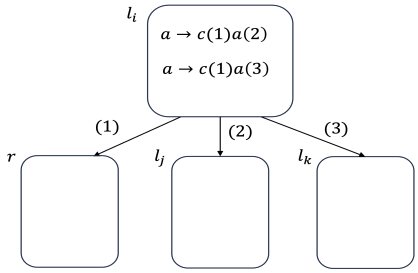



**Fig. 3.** An SNP-MC-CS system to simulate $l_i : (\text{ADD}(r), l_j, l_k)$.

– Simulation of the instruction $l_i : (\text{SUB}(r), l_j, l_k)$
The SUB instruction can be simulated by following the scheme of the figure 4. Upon receiving a spike $a$, the neuron $l_i$ consumes it and it sends a spike $d$ to the neuron $r$ that represents the register $r$. In the neuron $r$, the subtraction is carried out by the proposed rules. Observe that the first rule is applied only if the contents of the register is greater than zero, while the second rule is reserved for the zero case. A spike $a$ is sent to activate the following instruction $l_j$, or $l_k$.
Now, we must analize the case where at least two SUB instructions operate over the same register $r$. In such a case, additional rules and colored spikes are needed. For example, let us consider that, in the register machine program, the instructions $l_i : (\text{SUB}(r), l_j, l_k)$ and $l_m : (\text{SUB}(r), l_n, l_p)$ are used. Then,

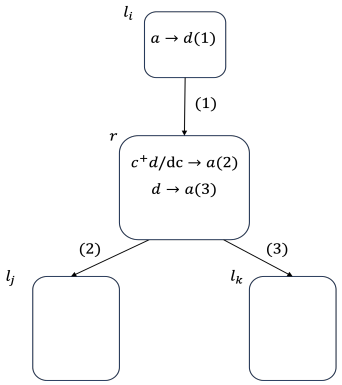



**Fig. 4.** An SNP-MC-CS system to simulate the instruction $l_i : (\mathrm{SUB}(r), l_j, l_k)$.

we can apply the scheme proposed in the figure 5. In such a case, we use different spikes to differentiate the two instructions: the spike $d$ is used to activate the first SUB instruction, while the spike $e$ is used to activate the other SUB instruction.
The scheme of the figure 5 can be enlarged as many times as necessary to simulate different SUB instructions applied to the same register.

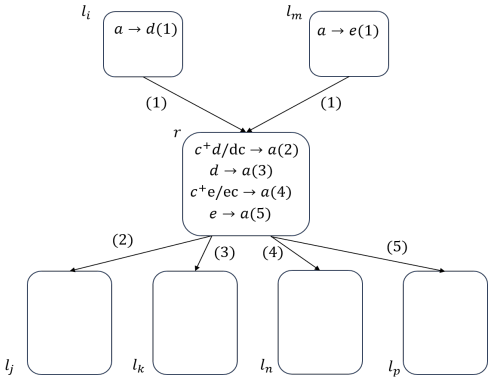



**Fig. 5.** An SNP-MC-CS system to simulate two instructions $l_i : (\mathrm{SUB}(r), l_j, l_k)$ and $l_m : (\mathrm{SUB}(r), l_n, l_p)$.

– Simulation of a HALT instruction
The simulation of a HALT instruction can be carried out by following the scheme of the figure 6. Upon receiving a spike $a$, the neuron $l_h$ sends a spike $h$ to the output register $r$ whose contents is the output of the simulated

register machine. The neuron $r$ receives the activating spike, and it consumes sequentially all the spikes $c$ that are used to represent the value of the register. For every consumed spike, a copy is sent out to the environment. Finally, the forgetting rule consumes the remaining spike $h$ and the computation halts since there is no spike that could activate another instruction. Observe that the output is obtained in a halting mode

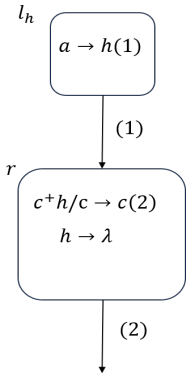



**Fig. 6.** An SNP-MC-CS system to simulate the HALT instruction.

## 4 Simulating other computation models by SNP-MC-CS systems.

Once we have proposed a new SN P system model, and we have shown its completeness, we will see in this section how the proposed model is able to simulate other ones that have been proposed in the field of membrane computing. In this way, the SNP-MC-CS model aims to be a unifying framework for other alternative models.

In the previous section, we have already seen that the model generalizes to models with multiple channels (SNP-MC systems), and, also, to models with multiple types of spikes (SNP-CS systems). Now, we will see how the model simulates virus machines [5] and neural-like P systems with plasmids [4].

### 4.1 Virus machines

Virus machines were first introduced in [37], and they have been used to compute recursive functions [31]. They have been proposed to generate, compute or recognize complex sets [29] and, recently, some characterizations of the number sets generated by the model have been studied [30].

In order to propose a simulation for virus machines by SNP-MC-CS systems, we provide a basic definition about the model as follows.

**Definition 6.** [5] A virus machine (VM, for short) of degree $(p, q)$, with $p, q \geq 1$, is a tuple $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$ where:

1. $\Gamma = \{v\}$ is a singleton alphabet;
2. $H = \{h_1, \ldots, h_p\}$ and $I = \{i_1, \ldots, i_q\}$ are ordered sets such that $v \notin H \cup I$ and $H \cap I = \emptyset$;
3. $D_H = (H \cup \{h_{out}\}, E_H, w_H)$ is a weighted directed graph, where $E_H \subseteq H \times (H \cup \{h_{out}\})$, $(h, h) \notin E_h$ for each $h \in H$, *out-degree*$(h_{out}) = 0$, and $w_H$ is a mapping from $E_H$ onto $\mathbb{N}\backslash\{0\}$ (the set of positive integer numbers);
4. $D_I = (I, E_I, w_I)$ is a weighted directed graph, where $E_I \subseteq I \times I$, $w_I$ is a mapping from $E_I$ onto $\mathbb{N}\backslash\{0\}$ and, for each vertex $i_j \in I$, the out-degree of $i_j$ is less than or equal to 2;
5. $G_C = (V_C, E_C)$ is an undirected bipartite graph, where $V_C = I \cup E_H$, being $\{I, E_H\}$ the partition associated with it. In addition, for each vertex $i_j \in I$, the degree of $i_j$ in $G_C$ is less than or equal to 1.
6. $n_j \in \mathbb{N}$ $(1 \leq j \leq p)$ and $i_1 \in I$
7. $h_{out} \notin I \cup \{v\}$ and $h_{out}$ is denoted by $h_0$ in the case that $h_{out} \notin H$.

□

In the figure 7, we can see the structure of a virus machine.

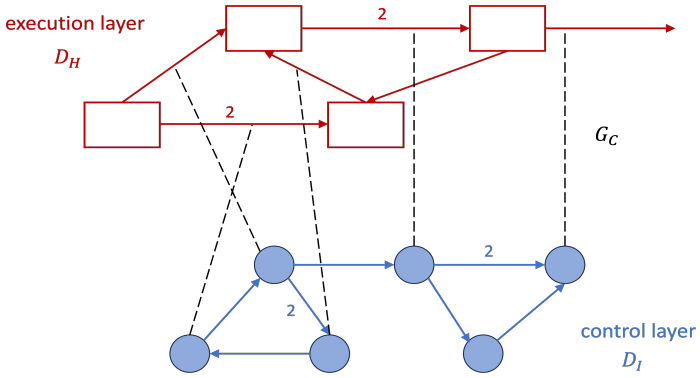



**Fig. 7.** The structure of a virus machine. In the top side we have an execution layer, in the bottom side we have a control layer, and both layers are connected by control executions.

We explain how a virus machine works: The nodes of the upper graph $D_H$ represent the hosts that contain viruses. The transmission and replication of viruses is done through the connections between these nodes. The weights of these connections represent how many copies of the virus are transmitted to the

receiving host (i.e., the weight $w_{s,t}$ means that whenever the host $s$ sends a copy of the virus to the host $t$, it replicates $w_{s,t}$ viruses in $t$). The connections between the nodes from $D_H$ are transmission channels that can be activated depending on the configuration in the $D_I$ graph, which is a graph of instructions. Each node in the $D_I$ graph activates the transmission channel of the $D_H$ graph depending on the connections that are specified in the $G_C$ graph. Once an instruction from $D_I$ has been executed, the next instruction to be executed will depend on whether viruses have been transmitted in the $D_H$ graph, and it will also depend on the out-degree of the last executed instruction. If there is more than one possible following instruction, provided that weights that connect the instructions are equal, the next instruction will be chosen non-deterministically. Otherwise, the next instruction will be selected according to the highest weight (if the viruses have been transmitted) or the smallest weight (if the viruses have not been transmitted). If there is no following instruction, then the next instruction will be interpreted as a halting instruction. The output of the system will be the number of viruses collected in the output host of the $D_H$ graph at the halting configuration, or in the environment if this had been predefined. For a greater level of detail on virus machines, we suggest the reader consult [5].

It has been proved that virus machines are a universal model of computation given that they can simulate the register machines [5].

Now, we provide a result that allows VM simulation by SNP-MC-CS systems.

**Theorem 2.** For every virus machine there exists an equivalent SNP-MC-CS system.

*Proof.* Let $\Pi = (\Gamma, H, I, D_H, D_I, G_C, n_1, \ldots, n_p, i_1, h_{out})$ be a VM. We can apply the following steps to obtain an equivalent SNP-MC-CS system:

1. Create a spike $v$ that represents a virus and three auxiliary spikes $s, f$ and $a_i$.
2. For all $h_i \in H$ create a neuron $h_i$
3. For all $i_j \in I$ create a neuron $i_j$
4. For each edge $(h_s, h_t) \in E_H$ associated with an instruction $i_k$:
   (a) Create a channel $c_{h_s,h_t}$ that represents the communication channel between hosts $h_s$ and $h_t$.
   (b) Create a channel $c_{i_k}$ that represents the control relationship between $(h_s, h_t)$ and the instruction $i_k$.
   (c) Add a synapse from neuron $h_s$ to neuron $h_t$ within a channel $c_{h_s,h_t}$.
   (d) Add a synapse from neuron $h_s$ to neuron $i_k$ within channel $c_{i_k}$, and another one in the opposite direction.
   (e) Create a spike $a_{h_s,h_t}$ that represents the activation signal for the channel $c_{h_s,h_t}$.
   (f) Add to the neuron $h_s$ the firing rules of the form:

$$a_{h_s,h_t} v^+ / v a_{h_s,h_t} \to v^{w_{h_s,h_t}}(c_{h_s,h_t}) s(c_{i_k})$$

This rule means that if an activation signal is received and there are viruses in the host, it consumes a virus and the activation signal, and it sends $w_{h_s,h_t}$ virus spikes to neuron $h_t$ and a spike $s$ to the instruction that triggered the transmission channel.

$$a_{h_s,h_t}/a_{h_s,h_t} \rightarrow f(c_{i_k})$$

This rule means that if an activation spike is received and there are no viruses in the host, it consumes the activation signal spike and it sends a spike $f$ to the instruction that triggered the transmission channel.

5. For each instruction $i_k$:

   (a) If the output degree of $i_k$ is equal to 2 then there are two different instructions $i_{k_1}$and $i_{k_2}$ such that $(i_k, i_{k_1}) \in E_I$ and $(i_k, i_{k_2}) \in E_I$.
   If the instruction $i_k$ is connected to a channel $(h_s, h_t)$:
   - Create a rule $a_i/a_i \rightarrow a_{h_s,h_t}(c_{i_k})$, which upon receiving an activation spike instruction $a_i$, consumes it and sends an activation spike to the channel.
   - Create two channels $c_{i_k,i_{k_1}}$ and $c_{i_k,i_{k_2}}$
   - If $w_I(i_{k_1}) = w_I(i_{k_2})$ then create the following firing rules:

$$s/s \rightarrow a_i(c_{i_k,i_{k_1}})$$

$$f/f \rightarrow a_i(c_{i_k,i_{k_1}})$$

$$s/s \rightarrow a_i\left(c_{i_k,i_{k_2}}\right)$$

$$f/f \rightarrow a_i(c_{i_k,i_{k_2}})$$

According to the previous rules, the following instruction is non-deterministically selected.
Otherwise, we take

$$i_{max} = \text{argmax}_{i\in\{i_{k_1},i_{k_2}\}} w_I((i_k, i))$$

$$i_{min} = \text{argmin}_{i\in\{i_{k_1},i_{k_2}\}} w_I((i_k, i)).$$

The following two firing rules are created:

$$s/s \rightarrow a_i(c_{i_k,i_{max}})$$

$$f/f \rightarrow a_i(c_{i_k,i_{min}})$$

So, if there was a virus in the host, the next instruction is selected according to the maximal weight channel and, otherwise, it is selected according to the minimal weight channel.
If the instruction $i_k$ is not connected to any channel:
- Create two channels $c_{i_k,i_{k_1}}$ and $c_{i_k,i_{k_2}}$

– Create the following firing rules:

$$a_i/a_i \to a_i(c_{i_k,i_{k_1}})$$
$$a_i/a_i \to a_i(c_{i_k,i_{k_2}})$$

So, the next instruction is non-deterministically selected.

(b) If the output degree of $i_k$ equals to one, and $(i_k, i_{k'}) \in E_I$, a rule of the form $a_i/a_i \to a_i(c_{i_k,i_{k'}})$ is created.

(c) If the output degree of $i_k$ is equal to 0, a forgetting rule $a_i \to \lambda$ is created.

In the proposed construction, the structure of the graphs in the virus machine is virtually simulated through the different communication channels. Furthermore, the dynamics of virus transmission, the activation of control instructions, and the opening of transmission channels based on those instructions are modeled by the different firing rules that we have defined.

We can formally prove that the complete simulation of a virus machine computation can be carried out on the SNP-MC-CS system.

□

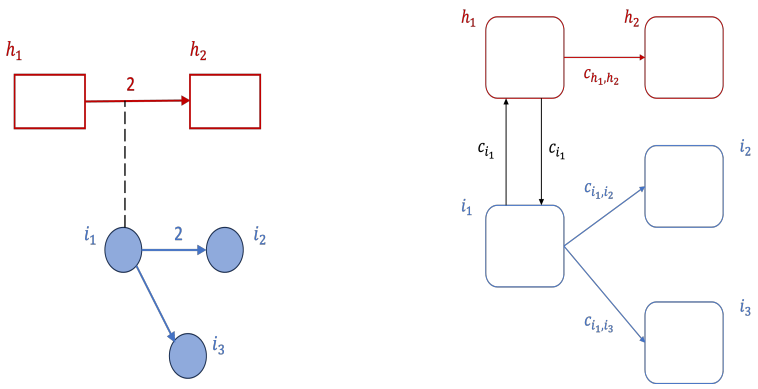



**Fig. 8.** A virus machine (left) and its equivalent SNP-MC-CS system (right).

*Example 1.* In the figure 8, we can see a very simple virus machine with two host nodes, and three instruction nodes. In the same figure, we can see the equivalent SNP-MC-CS. The rules in every neuron are the following:

– Neuron $h_1$
  - $a_{h_1,h_2}V^+/v\ a_{h_1,h_2} \to v^2(c_{h_1,h_2}), s(c_{i_1})$
  - $a_{h_1,h_2} \to f(c_{i_1})$

– Neuron $i_1$
  - $a_i \to a_{h_1,h_2}(c_{i_1})$
  - $s \to a_i(c_{i_1,i_2})$
  - $f \to a_i(c_{i_1,i_3})$

### 4.2 Neural-like P systems with plasmids

Neural-like P systems with plasmids were proposed in [4] inspired by the bacterial conjugation of genetic plasmids. In that work, the plasmids are objects of the system. This is a different approach to the recently computing by plasmids model [33], where the plasmids are finite sets of rules that can be moved in the region space of a P system. The use of multiple channels within neural-like P systems with plasmids has also been proposed [13].

In this section, we propose a simulation of neural-like P systems with plasmids by SNP-MC-CS systems. First we provide a basic definition of the model as follows.

**Definition 7.**[4] A neural-like P systems with plasmids (NP P system, for short) of degree $m \geq 1$, is a construct of the form $\Pi = (O, b_1, \ldots, b_m, link, in, out)$ where:

1. $O = \{p_1, \ldots, p_w\}$ is an alphabet of *plasmids*
2. $b_1, \ldots, b_m$ are bacteria of the form $b_i = (N_i, R_i)$ for $1 \leq i \leq m$ where $N_i = \langle n_1, n_2, \ldots, n_w \rangle$ is a vector and $n_j \geq 0$ is the initial number of plasmids $p_j$ inside bacterium $b_i$ for $1 \leq j \leq w$; and $R_i$ is a finite non-empty set of plasmid rules of the form $C/r_1, r_2, \ldots, r_n$ where condition $C$ is a multiset over $O$, and, each $r_k$ for $1 \leq k \leq n$ is either one of two forms:
   (a) A *transmit* operation: $P \rightarrow out$, where $P$ is a submultiset of $C$;
   (b) A *kill* operation: $P \rightarrow \lambda$, where $P$ is a submultiset of $C$;
3. $link \subset \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin link$ for $1 \leq i \leq m$ (the links between bacteria);
4. $in, out$ indicates the input and output bacterium of the system, respectively. They can be omitted, depending on whether the system is generating outputs or accepting inputs.

□

We explain how a NP P system works. At every time, the system executes a rule in each bacterium (whenever possible), this rule will be chosen in a non-deterministic way among the applicable rules. A rule $C/r_1, r_2, \ldots, r_n$ in bacteria $b_i$ is applicable if at least each plasmid in $C$ is in $b_i$. When a rule is executed, all its operations are executed sequentially starting from $r_1$ up to $r_n$. The transmit operations ($P \rightarrow out$ ) send a copy of every plasmid in $P$ to every linked bacterium $b_j$, that is $(i, j) \in link$. On the other hand, the kill operations ($P \rightarrow \lambda$) remove all the plasmids defined in $P$. Observe that the transmit operations do not remove any plasmid in the bacteria. The computation halts whenever no bacteria can apply a rule.

It has been proved that NP P systems are a universal model of computation given that they can simulate the register machines [4].

The NP P systems with multiple channels (NPMC P systems) [13] have been proved to be universal, and they can achieve universality by using less computational resources than NP P systems (the number of plasmids and rules decreases).

In the following, we will prove the equivalence between NP P systems and SNP-MC-CS systems. The proof can be easily adapted to NPMC P systems.

**Theorem 3.** For every NP P system there exists an equivalent SNP-MC-CS system.

*Proof.* Let $\Pi = (O, b_1, \ldots, b_m, link, in, out)$ be an NP P system. We can apply the following steps to obtain an equivalent SNP-MC-CS system:

1. Create three spikes $a_p, s_p$ and $h_p$.
2. Create three channels $c_h, c_p$ and $c_c$.
3. Create the neuron $n_{halt}$ with the rules $h_p^m \to h_p(c_h)$ and $h_p \to \lambda$.
4. For each $p_i \in O$ define a spike $p_i$.
5. For each bacterium $b_i = (N_i, R_i)$ $(1 \leq i \leq m)$:
   (a) Create three neurons: $b_i$, $c_{b_i}$ and $g_{b_i}$. Initially, every neuron has a spike $a_p$. The neuron $c_{b_i}$ also has a spike $s_p$.
   (b) Add the synapses $(b_i, c_{b_i})$, $(c_{b_i}, b_i)$, $(c_{b_i}, g_{b_i})$ and $(g_{b_i}, c_{b_i})$ connected by the channel $c_c$.
   (c) Add the synapse $(c_{b_i}, n_{halt})$ connected by the channel $c_h$.
   (d) For all $n_j \in N_i$ add $n_j$ spikes of type $p_j$ to the neuron $b_i$.
   (e) Add the rule $a_p \to a_p(c_c)$ to the neuron $g_{b_i}$.
   (f) Add the rules $a_p s_p \to a_p(c_c)$, and $a_p \to a_p(c_c)h_p(c_h)$ to the neuron $c_{b_i}$.
   (g) For every rule of the form $C/r_1, r_2, \ldots, r_n \in R_i$ define a rule such that:
      i. The regular expression is defined as $c_1 c_2 \ldots c_n p_1^* p_2^* \ldots p_w^*$ where $c_1 c_2 \ldots c_n$ is the multiset $C$.
      ii. The spikes that consume the rule is the sum of all the multisets of plasmids consumed by the kill operations plus the spike $a_p$.
      iii. The right-hand side of the rule is defined by the multiset resulting from adding all the multisets of the transmit operations by the channel $c_p$, and the spike $s_p$ by the channel $c_c$.
   (h) Add the rule $a_p \to \lambda$ to the neuron $b_i$.
6. For each $(i, j) \in link$ add a synapse $(b_i, b_j)$ connected by the channel $c_p$.
7. The input neuron is defined by $in$.
8. Given the output bacteria $out$, add the synapses $(n_{halt}, b_{out})$ and $(b_{out}, out)$ connected by the channel $c_h$.
9. Add to the neuron $out$ the rules $h_p p_1^* p_2^* \ldots p_w^*/p_i \to p_i(c_h)$ for all $1 \leq i \leq w$ and $h_p \to \lambda$

The system works as follows: As only one rule can be applied to each bacterium at a time, there is a single spike $a_p$ in the neuron representing the bacterium, and all the rules consume it; therefore, once one rule is applied, the others can no longer be applied.

To maintain a constant number of $a_p$ spikes per bacterium, each bacterium has a control neuron $c_{b_i}$ and a generation neuron $g_{b_i}$ which, at each execution step, send a spike $a_p$ to the bacterium $b_i$. To prevent $a_p$ spikes from accumulating in the bacterium if it does not execute a rule, a forgetting rule $a_p \to \lambda$ is added. However, this presents a new problem: when the NP P system reaches a halting

state (no bacterium is applying any rules), the system as a whole will not reach a halting state, because the generators continue to apply rules.

To solve this, we have the spikes $s_p$, $h_p$ and the neuron $n_{halt}$. All control neurons are connected to the neuron $n_{halt}$; when a rule is applied to a bacterium, a spike $s_p$ is sent to its control neuron. Therefore, if that neuron does not receive a spike $s_p$ at a specific point in time, this means that its bacterium has not applied any rules, so it sends a spike $h_p$ to the neuron $n_{halt}$.

If, at a given moment, the neuron $n_{halt}$ receives a number of $h_p$ spikes smaller than the number of bacteria in the system, this means that not all of them have stopped, so it applies a forgetting rule and everything continues to function normally. However, if, on the other hand, the number of spikes $h_p$ received is equal to the number of bacteria in the system, this means that all of them have stopped; consequently, it sends a spike $h_p$ to all the control neurons, which blocks them and causes them to stop generating spikes, thereby reaching the halting state.

We can formally prove that the complete simulation of a NP P system with plasmids computation can be carried out by the SNP-MC-CS system.

□

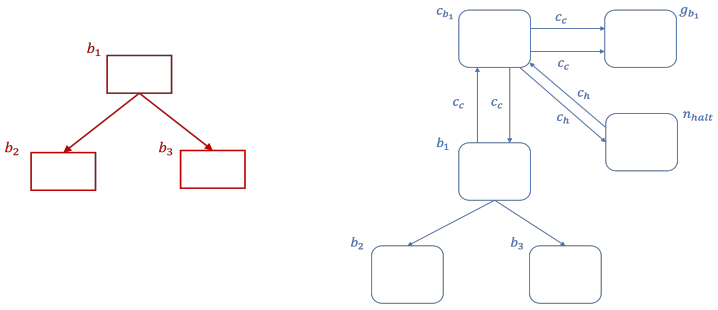



**Fig. 9.** A neural-like P system with plasmids (left) and its equivalent SNP-MC-CS system (right).

*Example 2.* In the figure 9, we can see a very simple neural-like P system with three bacteria. In the same figure, we can see the equivalent SNP-MC-CS.

The rules in the bacteria $b_1$ are the following:

- $p_1p_2/p_2 \to out,\ p_1 \to \lambda$
- $p_1p_3/p_3 \to out,\ p_1 \to \lambda$

The rules in the main neurons of the SNP-MC-CS system are the followings:

- Neuron $c_{b_1}$ with the spikes $a_p s_p$

- $a_p s_p \to a_p(c_c)$
- $a_p \to a_p(c_c), h_p(c_h)$

– Neuron $g_{b_1}$ with the spike $a_p$

- $a_p \to a_p(c_c)$

– Neuron $b_1$ with the spike $a_p$

- $p_1 p_2 p_1^* p_2^* p_3^* / p_1 a_p \to p_2(c_p), s_p(c_c)$
- $p_1 p_3 p_1^* p_2^* p_3^* / p_1 a_p \to p_3(c_p), s_p(c_c)$
- $a_p \to \lambda$

– Neuron $n_{halt}$

- $h_p^3 \to h_p(c_h)$
- $h_p \to \lambda$

## 5 Conclusions

In this work we have presented a new model that combines two previously proposed features of Spiking Neural P systems: the use of multiple channels in the rules and in the network topology, and the use of a multi-spike alphabet. Initially, we demonstrated the completeness of the new model through simulations of register machines. This is not a novel aspect, since each variant used separately already exhibited this universality property. However, in our proposal, we present new simulations that incorporate both characteristics of the model.

Subsequently, we have demonstrated that the proposed model is capable of directly simulating other computational models of interest in the field of membrane computing: virus machines and neural-like P systems with plasmids. In both cases, we have proposed a direct construction of the model from the definitions of VM and NP P systems.

In section 3.1, we have provided different schemes for simulating register machine programs. However, the simulation is fundamentally sequential. In future works, we will explore the ability of this model to simulate other intrinsically parallel models, such as parallel random access machines (PRAM) [7].

Furthermore, as we mentioned in the introduction to this work, our proposal can be formulated as a general-purpose computing environment in which different neural networks must coexist, cooperate and influence one another, and in which neurons can specialise in different types of information. This general framework will be explored in future works.

**Acknowledgments.** This work has received funding from the Generalitat Valenciana within the Prometeo program in the project *"A disruptive way to ameliorate the diagnosis and treatment of sensorineural diseases."* CIPROM/2023/026.

## References


1. A. Alhazov, R. Freund, M. Oswald, M. Slavkovik (2006) Extended Spiking Neural P Systems. In: H.J. Hoogeboom, G. Păun, G. Rozenberg, A. Salomaa (eds). Membrane Computing. WMC 2006. LNCS vol, 4361. pp 123-134 Springer.
2. F.G.C. Cabarle (2024) Thinking about spiking neural P systems: some theories, tools, and research topics *Journal of Membrane Computing* 6 pp 148-167.
3. F. G. C. Cabarle, H. N. Adorna, M. J. Pérez-Jiménez, T. Song (2015) Spiking neural P systems with structural plasticity. *Neural Computing and Applications* 26 pp 1905–1917.
4. F.G.C. Cabarle, X. Zeng, N. Murphy, T. Song, A. Rodríguez-Patón, X. Liu. (2021) Neural-like P systems with plasmids. *Information and Computation*, Vol. 281, 194776.
5. X. Chen, M.J. Pérez-Jiménez, L. Valencia-Cabrera, B. Wang, X. Zeng (2016) Computing with viruses. *Theoretical Computer Science* 623 pp 146-159.
6. S. Fan, P Paul, T. Wu, H. Rong, G. Zhang (2020) On Applications of Spiking Neural P systems. *Applied Sciences* 10(20) 7011.
7. S. Fortune, J. Wyllie (1978) Parallelism in random access machines In Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, pp. 114-118
8. S. Ghosh-Dastidar, H. Adeli (2009) Spiking Neural Networks. *International Journal of Neural Systems* 19(4) pp 295-308.
9. T.O. Ishdorj, A. Leporati, L. Pan, J. Wang (2009) Solving NP-Complete Problems by Spiking Neural P Systems with Budding Rules. Proceedings of the *10th International Workshop on Membrane Computing*, LNCS Vol. 5957 pp 335–353. Springer.
10. M. Ionescu, Gh. Păun, T. Yokomori (2006) Spiking neural P systems. *Fundamenta Informaticae*, 71 pp 279-308
11. L. Kari, G. Rozenberg (2008) The Many Facets of Natural Computing. *Communications of the ACM*, 51(10) pp 72-83.
12. A. Leporati, G. Mauri, C. Zandron (2022) Spiking neural P systems: main ideas and results. *Natural Computing* Vol. 21, pp 629-649.
13. Y. Li, B. Song , X. Zeng (2023) Neural-Like P Systems With Plasmids and Multiple Channels. *IEEE Transactions on Nanobioscience*, Vol. 22, No, 2 pp 420-429
14. Z. Lv, Q.Yang, H. Peng, X. Song and J. Wang Z. (2021) Computational power of sequential spiking neural P systems with multiple channels. *Journal of Membrane Computing* Vol. 3, Iss. 4 pp 270–283
15. C. Martín-Vide, Gh. Păun, J. Pazos, A. Rodríguez-Patón (2003) Tissue P systems. *Theoretical Computer Science* 296(2) pp 295-326.
16. Q. Miao, Y. Shen, W. Zang, Y. Zhao (2025) Multiple channels spiking neural P systems with lateral inhibition, rules dynamic generation and removal. *Theoretical Computer Science* Vol. 1042, 115236.
17. M.L. Minsky (1967) *Computation: Finite and Infinite Machines* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
18. X. Ning. G. Yang, Z. Sun, X. Song (2024) On the Universality of Spiking Neural P Systems With Multiple Channels and Autapses *IEEE Access* Vol. 12, pp 8773-8779.
19. L. Pan, Gh. Păun, M.J. Pérez-Jiménez (2011) Spiking neural P systems with neuron division and budding. *Science China Information Sciences* 54 pp 1596–1607.
20. L. Pan, J. Wang, H.J. Hoogeboom (2012) Spiking Neural P Systems with Astrocytes. *Neural Computation* 24 (2012) pp 805-825.
21. L. Pan, X. Zeng, X. Zhang (2011) Time-free Spiking Neural P Systems. *Neural Computation* 23(5) pp 1320–1342.

22. P. Paul, P. Sosík (2024) Solving the SAT problem using spiking neural P systems with coloured spikes and division rules *Journal of Membrane Computing* Vol. 6, Issue 3, pp 222-233.
23. Gh. Păun (2000) Computing with Membranes. *Journal of Computer and Systems Sciences* 61, pp 108-143.
24. Gh. Păun (2002) *Membrane Computing. An Introduction.* Springer.
25. Gh. Păun, Gr., Rozenberg, A. Salomaa, A. (eds.) (2010) The Oxford Handbook of Membrane Computing. Oxford University Press, NY
26. H. Peng, J. Wang (2024) Advanced Spiking Neural P Systems. Models and Applications. Springer.
27. H. Peng, J. Yang, J. Wang, T. Wang, Z. Sun, X. Song, X. Lou. (2017) Spiking neural P systems with multiple channels. *Neural Networks,* Vol. 95, pp 66-71.
28. H. Peng, J. Wang, M.J. Pérez-Jiménez, H. Wang, J. Shao, T. Wang (2013) Fuzzy reasoning spiking neural P system for fault diagnosis. *Information Sciences* 235 pp 106–116.
29. A. Ramírez de Arellano, D. Orellana Martín M.J. Pérez Jiménez (2023) Generating, computing and recognizing with virus machines. *Theoretical Computer Science* Vol. 972 114077.
30. A. Ramírez de Arellano, F.G.C. Cabarle, D. Orellana Martín M.J. Pérez Jiménez (2026) On the normal forms and computational power of virus machines *Theoretical Computer Science* Vol. 1083 116139.
31. Á. Romero-Jiménez, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez (2015) Computing partial recursive functions by virus machines Proceedings of the 16th International Conference on Membrane Computing (CMC16). Gr. Rozenberg, A. Salomaa, J.M. Sempere, C. Zandron (eds.). LNCS Vol. 9504 pp. 353-384 Springer.
32. G. Rozenberg, A. Salomaa, Eds. (1997) *Handbook of Formal Languages.* Vols. I, II and III. Springer-Verlag.
33. J.M. Sempere (2026): Computing by plasmids. *Journal of Membrane Computing*, Online first.
34. T. Song, A. Rodríguez-Patón, P. Zheng, X. Zeng (2018) Spiking Neural P Systems With Colored Spikes. *IEEE Transactions on Cognitive and Developmental Systems* 10(4) pp 1106-1115.
35. X. Song, L. Valencia-Cabrera, H. Peng, J. Wang (2021) Spiking neural P systems with autapses, *Information Sciences*, Vol. 570, pp 383-402,.
36. Z. Sun, Q. Wang, Z. Wei (2019) Fault location of distribution network with distributed generations using electrical synaptic transmission-based spiking neural P systems. *International Journal of Parallel, Emergent and Distributed Systems* 36(1) pp 11-27.
37. L Valencia-Cabrera, MJ Pérez-Jiménez, X Chen, B Wang, X Zeng (2015) Basic Virus Machines. In Proceedings of the 16th International Conference on Membrane Computing (CMC16), José M. Sempere and Claudio Zandron (eds.) pp 323-342.
38. S. Verlan, R. Freund, A. Alhazov, S. Ivanov, L. Pan (2020) A formal framework for spiking neural P systems. *Journal of Membrane Computing* Vol. 2, pp 355–368.
39. G. Zhang, H. Rong, F. Neri, M.J. Pérez-Jiménez (2014) An optimization spiking neural P system for approximately solving combinatorial optimization problems. *International Journal of Neural Systems* 24(5) 1440006.
40. G. Zhang, S. Verlan, T. Wu, F.G.C. Cabarle, J. Xue. D. Orellana-Martín, J. Dong, L. Valencia-Cabrera, M.J. Pérez-Jiménez (2024) Spiking Neural P Systems. Theory, Applications and Implementations. Springer.

# Numerical 2D P Colonies as a Data Management Architecture

Daniel Valenta[2] [0009−0005−0781−7755] and
Miroslav Langer[1] [0000−0001−5990−7780]

[1] Silesian University in Opava daniel.valenta@fpf.slu.cz
[2] VSB - Technical University of Ostrava miroslav.langer@vsb.cz

**Abstract.** Biological computing has developed from theoretical models into a broad research field including DNA computing, molecular computing, membrane computing, synthetic biology, and living cellular computers. Most existing approaches focus on the mechanisms by which biological or bio-inspired systems perform computation. Considerably less attention has been devoted to the question of how data should be represented, stored, queried, indexed, and updated inside such computational architectures. Unlike biological databases storing genomic or biomedical information in conventional database systems, this paper addresses the different problem of an internal data management layer for biological computers. We propose numerical 2D P colonies as a data management architecture capable of representing arbitrary information by numerical encoding and manipulating it by means of simple parallel agents. The proposed architecture adopts principles of relational database systems and maps them onto the environment of a numerical 2D P colony. We describe the organization of the environment into queue, data, index, log, and result areas, and we outline the basic data manipulation operations INSERT, UPDATE, DELETE, and SELECT. The aim of the paper is to introduce a formal bio-inspired storage and query-processing layer that may support future biological computing architectures.

**Keywords:** 2D P Colony · Membrane Computing · Data Management Architecture.

## 1 Introduction

During the last three decades, biological computing has evolved from theoretical models to experimentally realizable computing substrates, including DNA computing, molecular computing, membrane computing, synthetic biology, and living cellular computers. Despite substantial progress in computational models, considerably less attention has been devoted to the problem of systematic data management. While conventional computing relies on mature data management architectures such as relational database management systems, biological computing currently lacks a general-purpose architecture for persistent data representation, querying, indexing, and concurrent data manipulation. Existing

research primarily focuses on computational mechanisms rather than on the organization and management of data processed by biological computers.

This distinction is important. In classical computer systems, computation is not considered in isolation. Processors, memory hierarchies, file systems, database systems, transaction mechanisms, and query processors form an integrated architecture. Database systems provide a stable abstraction for persistent data and allow applications to store, search, update, and combine information without directly controlling the physical storage medium. If biological computers are expected to process complex and heterogeneous information, an analogous data management layer becomes a natural requirement.

The present paper proposes such a layer using numerical 2D P colonies. A 2D P colony is a bio-inspired multi-agent computational model in which simple agents live and move in a shared two-dimensional environment. Agents are equipped with small programs consisting of evolution, communication, and motion rules. Numerical extensions of 2D P colonies make it possible to represent and manipulate numerical values, which provides a uniform abstraction for encoded data. Since arbitrary information can be encoded numerically, the model can serve as a substrate for storing and processing heterogeneous data in a common form. The term data in this paper does not refer specifically to biological datasets such as genomic, proteomic, transcriptomic, or clinical data. It denotes arbitrary information processed by a biological computer.

The proposed architecture is inspired by relational database management systems. Rows, attributes, indexes, logs, queues, and query results are represented as objects placed in different areas of the two-dimensional environment. Agents perform operations analogous to INSERT, UPDATE, DELETE, and SELECT. The inherent parallelism of the colony model allows several agents to work simultaneously, while the restriction that two agents cannot communicate with the same environmental object in the same computational step gives a natural basis for conflict avoidance.

## 2 Related Work

### 2.1 Biological and Molecular Computing

Biological computing is a broad area covering theoretical and experimental approaches in which computation is inspired by, modeled after, or physically implemented in biological substrates. DNA computing demonstrated that molecular structures can be used to solve computational problems by exploiting massive parallelism at the molecular level [1]. Later work on DNA strand displacement, molecular circuits, and chemical reaction networks further developed this idea by treating molecular interactions as programmable information-processing mechanisms.

Synthetic biology provides another important line of research. Genetic circuits, regulatory networks, and engineered cellular systems can implement logical functions, memory-like behavior, and signal processing inside living cells. These

systems are often designed as biological analogues of electronic circuits, but their data representation is usually tightly coupled to specific molecular species, regulatory mechanisms, or experimental settings.

Living cellular computers and wetware systems represent a more recent direction in which living cells, neural cultures, or organoids are treated as computational substrates. These approaches emphasize adaptability, embodiment, learning, and interaction with biological environments. However, they also highlight the need for more systematic abstractions that make biological computation programmable, composable, and reliable.

Across these approaches, the primary emphasis is usually placed on computation itself: how to implement logical functions, solve optimization problems, process signals, or simulate computational models. The question of a general-purpose internal data management architecture has received much less attention. In this paper, we therefore focus not on the biological implementation of a particular computing device, but on the formal organization of data that such a device may need to manipulate.

### 2.2 Membrane Computing and P Colonies

Membrane computing, introduced by Păun, studies computational models inspired by the structure and functioning of living cells [8–10]. P systems use multisets of objects, membranes, and evolution rules to describe massively parallel computations. Many variants have been introduced, including cell-like, tissue-like, neural-like, and numerical P systems.

P colonies are a simple class of membrane systems composed of agents living in a shared environment [5]. Each agent contains a small number of objects and a set of simple programs. The agents interact with the environment by evolution and communication rules. Despite their simplicity, several variants of P colonies have been shown to be computationaly complete [4, 3].

A 2D P colony extends this idea by placing the environment on a two-dimensional grid. Agents can move in the environment and interact with objects located at particular positions. This spatial interpretation makes the model suitable for simulations of multi-agent systems and spatial processes. Previous work introduced 2D P colonies, evolving environments, blackboard communication, numerical operations, and simulation tools [2, 6, 11, 7].

Numerical 2D P colonies are especially relevant for the present paper. By allowing numerical values to be stored and manipulated, they provide a uniform representation layer. This is crucial for the proposed data management architecture, because arbitrary data can be encoded as numbers and processed by agents that operate on numerical objects rather than on domain-specific structures.

## 3 Requirements for Data Management in Biological Computers

A data management architecture for biological computers should satisfy requirements analogous to those addressed by database systems in conventional comput-

ing. These requirements cannot be transferred mechanically, because biological and bio-inspired models have different primitives, different forms of parallelism, and different constraints. Nevertheless, several general requirements can be identified.

*Uniform data representation.* The architecture should represent heterogeneous information in a common form. Biological computers may process sequences, signals, sensor values, symbolic structures, or encoded knowledge. A numerical representation layer is therefore useful, because different data types can be mapped to numerical values and manipulated by the same class of agents.

*Persistent storage abstraction.* The system should distinguish data storage from transient computation. Objects that represent stored rows must remain in the environment until they are explicitly changed or deleted. This allows the colony to act not only as a computational process but also as a persistent information structure.

*Parallel access.* Biological and bio-inspired systems are naturally parallel. A suitable data management architecture should not serialize all operations into a single control flow. Instead, it should allow many agents to access different parts of the data simultaneously.

*Concurrent updates.* If several agents manipulate data at the same time, write conflicts and inconsistent states must be addressed. The architecture should provide mechanisms comparable to locking, atomic exchange, or isolation, but formulated using the primitives of the underlying colony model.

*Efficient search.* Sequential scanning is simple but does not scale. An index structure is required to support efficient lookup of rows according to keys. The index should be representable inside the same environment and maintainable by agents.

*Logging and recovery.* Database systems use logs to support recovery after failures. In a formal biological computing model, logging can be interpreted as an explicit record of operations that allows reconstruction of a previous configuration or rollback of incomplete updates.

*Query evaluation.* The architecture should support at least elementary query processing. In the present paper, we focus on selection and counting, using result objects deposited by agents in a dedicated result area. More complex relational operations, such as joins and aggregations, are left for future work.

## 4 Numerical 2D P Colony

We recall and adapt the formal definition of the numerical 2D P colony introduced in [11]; in particular, the blackboard component is omitted, and the

environmental function is reinterpreted to encode string-based data rather than a fitness value.

**Definition 1.** *A numerical 2D P colony is a construct*

$$\Pi = (V, e, Env, A_1, \ldots, A_k, f), \quad k \geq 1,$$

*where*

- $V$ *is the alphabet of the colony. The elements of the alphabet are called objects. Numerical objects are special objects that can contain an arbitrary number.*
- $Env$ *is a triplet* $(m \times n, w_E, f_E)$*, where* $m \times n$*,* $m, n \in N$ *is the size of the environment.* $w_E$ *is the initial contents of the environment, it is a matrix of size* $m \times n$ *of multisets of objects over* $V - \{e\} \cup \{f_E(x)\}$*.* $f_E$ *is an environmental function, where* $f_E(x)$ *encodes a combination of strings over an alphabet* $\Sigma$ *(like the English alphabet extended with digits) that could be stored as data, hence the domain and range of* $f_E$ *are given by the definition of the problem.*
- $A_i, 1 \leq i \leq k$*, are the agents. Each agent is a construct* $A_i = (o_i, P_i, [o, p])$*,* $0 \leq o \leq m, 0 \leq p \leq n$*, where*
  - $o_i$ *is a multiset over* $V$*, it determines the initial state (contents) of the agent,* $|o_i| = T_{max} + 1$*, where* $T_{max} > 1$ *is the maximum number of table columns. One additional object is reserved as a working object and does not store data. In practice,* $T_{\max}$ *can be determined dynamically or set to a fixed value (e.g.,* 32*), which is sufficient for most database tables.*
  - $P_i = \{p_{i,1}, \ldots, p_{i,l_i}\}$*,* $l \geq 1$*,* $1 \leq i \leq k$ *is a finite set of programs, where each program contains exactly 2 rules. Each rule is in the following form:*
    - $a \to b$*, the evolution rule,* $a, b \in V$*,*
    - $c \leftrightarrow d$*, the communication rule,* $c, d \in V$*,*
    - $[a_{q,r}] \to s$*,* $a, q, r \in V$*,* $0 \leq q, r \leq 2$*,* $s \in \{\Leftarrow, \Rightarrow, \Uparrow, \Downarrow\}$*, the motion rule,*
    - $a \rightsquigarrow x$*,* $x \in \mathbb{R}, a, x \in V$*, is the communication rule to read the numbers from the environment.*

    *If the program contains an evolution or communication rule* $r_1, r_2$ *that each works with objects with numbers, it can be extended by a condition:* $\langle x > y : r_1, r_2 \rangle$*,* $\langle x \geq y : r_1, r_2 \rangle$*,*
  - $[o, p]$*,* $0 \leq o \leq m, 0 \leq p \leq n$*, is an initial position of agent* $A_i$ *in the 2D environment.*
- $f \in V$ *is the final object of the colony.*

A configuration of the numerical 2D P colony is given by the state of the environment — a matrix of type $m \times n$ with pairs — multiset of objects over $V - \{e\}$, and a number — as its elements, and by the states of all agents — pairs of objects from the alphabet $V$, and the coordinates of the agents. An initial configuration is given by the definition of the numerical 2D P colony. In case of new storage, the environment contains only $e$, and the agents are located in position $[0, 0]$ $(Z_Q)$.

The environment of the numerical 2D P colony is, in general, given by the matrix. Each element of the matrix is represented by the multiset of objects over the alphabet $V - \{e\}$, and it also contains the value of the environmental function $f_E$, encoding a combination of strings over the alphabet $\Sigma$ that could be stored as data.

A computational step consists of three steps. In the first step, the set of the applicable programs is determined according to the current configuration of the numerical 2D P colony. In the second step, one program from the set is chosen for each agent, in such a way that there is no collision between the communication rules belonging to different programs. In the third step, chosen programs are executed, and the values of the environment are updated.

A change of the configuration is triggered by the execution of programs, and by updating values via functions. It involves changing the state of the environment, and the contents and placement of the agents.

A computation is non-deterministic and maximally parallel. The computation ends by halting, when there is no agent that has an applicable program.

The result of the computation is the number of copies of the final object placed in the environment at the end of the computation.

## 5 Proposed Architecture

The proposed architecture uses a numerical 2D P colony as a data management layer. The environment of the colony is interpreted as a structured storage space divided into several functional areas. Agents represent elementary execution units. Each agent carries a small number of objects encoding its state and the data currently processed. Database operations are represented as tasks placed into a queue area. Agents pick up tasks, move to the appropriate area, perform local interactions with environmental objects, and deposit results or follow-up tasks.

Let the environment be an $m \times n$ grid. Each position of the grid contains a multiset of objects over an alphabet $V$. The basic environmental object is denoted by $e$. Numerical values are represented by numerical objects or by objects carrying encoded numerical values. The interpretation of these numerical values depends on the encoding layer. For example, integers, floating-point numbers, strings, identifiers, or domain-specific biological signals can be mapped to numerical values before they are inserted into the colony.

The architecture divides the environment into five logical areas:

**Queue area $Z_Q$.** Stores pending operations. Each operation is encoded as a pair consisting of an operation object, such as `ins`, `upd`, `del`, or `sel`, and a data object carrying the values or keys required by the operation.

**Data area $Z_D$.** Stores table rows. Each row of the relational table is mapped to one row or block of cells in the data area. Columns are represented by positions inside the row.

**Index area $Z_I$.** Stores index entries. Each entry contains a key and a pointer to the corresponding row in the data area. The index can be implemented as a sorted list or as a tree-like structure.

**Log area $Z_L$.** Stores records of operations before they are applied to the data area. This provides a basis for recovery and for reasoning about consistency.

**Result area $Z_R$.** Stores query results. For counting queries, the number of copies of a final result object $f$ represents the result.

This decomposition is shown conceptually in Table 1. The partitioning is logical rather than physical in the database sense: all areas are still parts of the same two-dimensional colony environment.

**Table 1.** Logical areas of the proposed data management architecture.

| Area | Role | Database analogue |
|---|---|---|
| $Z_Q$ | pending tasks | transaction or operation queue |
| $Z_D$ | stored rows | data pages or heap file |
| $Z_I$ | search structures | index |
| $Z_L$ | operation history | log |
| $Z_R$ | query outputs | result buffer |

The architecture is intentionally minimal. It does not require agents to understand a complete query language. Instead, each agent executes a small program corresponding to one operation. More complex behavior emerges from the interaction of many simple agents, their movement in the environment, and the organization of objects into areas.

A central feature is the queue area. All agents may initially be identical or almost identical. Their role is determined when they pick up a task from $Z_Q$. The communication rule by which an agent obtains the operation object and the data object is atomic in the sense of the colony model. Therefore, two agents cannot pick up the same task in the same computational step. This gives a natural dispatching mechanism without requiring a centralized scheduler.

Another important feature is locality. An agent does not need a global view of the entire database. Once it has obtained its task, it moves according to encoded coordinates, keys, or pointers. This makes the architecture compatible with the spatial nature of 2D P colonies and supports parallelism: agents working on different rows or different index branches can operate simultaneously.

It is important to distinguish the proposed architecture from biologically inspired data storage technologies, such as DNA-based storage. Such approaches primarily address the physical representation, encoding, preservation, and retrieval of information using a biological substrate. These capabilities, however, do not by themselves constitute a general-purpose data management architecture providing abstractions for the organization, querying, indexing, concurrent manipulation, and management of stored data. The approach proposed in this

paper addresses this higher architectural level. Numerical 2D P colonies are used to organize encoded information and to provide mechanisms for its manipulation by interacting agents. Consequently, DNA storage and the proposed architecture address different abstraction levels rather than competing storage technologies. A biological storage medium, including DNA-based storage, could in principle provide the underlying physical representation of information, while the proposed architecture could provide a data management layer operating above such a representation.

## 6 Relational Storage Model

The relational model represents data as relations, usually implemented as tables. In the proposed architecture, a relation $R(A_1, A_2, \ldots, A_k)$ is represented by a region of the data area $Z_D$. Each tuple corresponds to one row in $Z_D$, and each attribute $A_j$ corresponds to a designated cell or group of cells in that row.

The term heterogeneous data used in this paper refers to heterogeneity at the input and semantic level, rather than to heterogeneity of the internal database model. Different types of information, such as numerical values, strings, sequences, identifiers, or signals, may have different original representations and semantics. Before being stored in the colony, these values are transformed by the encoding layer into a uniform numerical representation. The relational storage model therefore operates on homogeneous numerical objects, independently of the original type and meaning of the encoded information. Consequently, heterogeneity is handled by the encoding layer, whereas the organization of the encoded data within the colony remains relational.

Let a tuple be

$$t = (v_1, v_2, \ldots, v_k),$$

where each value $v_j$ is encoded as a numerical object. The row representing $t$ contains a row marker $r$ and the encoded values $v_1, \ldots, v_k$. Empty cells contain only the environmental object $e$. Deleted rows are marked by a deletion object $d$ and may later be reused by insert operations.

The basic object alphabet contains at least the following objects:

$$V_D = \{e, r, d, k, v, p, f, lk, ins, upd, del, sel, ok, ri\}.$$

The meanings of these objects are summarized in Table 2. In a numerical implementation, some of these objects may carry numerical parameters. For instance, $k$ may carry a key value, $v$ may carry an encoded attribute value, and $p$ may carry coordinates of a row in $Z_D$.

### 6.1 Index Representation

An index maps keys to row positions. The simplest index is a sorted list in $Z_I$, where each entry contains a key object and a pointer object. An agent searching for a key can move through the index according to a binary search strategy. If

**Table 2.** Basic objects used in the relational storage model.

| Object | Meaning |
|---|---|
| $e$ | basic environmental object |
| $r$ | marker of an existing row |
| $d$ | marker of a deleted row |
| $k$ | key object |
| $v$ | value object |
| $p$ | pointer to a row or cell |
| $f$ | final or result object |
| $lk$ | lock object |
| $ins$ | INSERT operation marker |
| $upd$ | UPDATE operation marker |
| $del$ | DELETE operation marker |
| $sel$ | SELECT operation marker |
| $ok$ | successful completion marker |
| $ri$ | re-indexing request |

$r_{max}$ is the maximum number of rows, then the number of visited index entries is $O(\log r_{max})$ in the abstract model.

A more advanced variant uses a B-tree-like layout. The index area is divided into bands representing levels of the tree. Cells inside a node alternate between key objects and child pointers. Agents navigate the structure by comparing the searched key with stored keys and following the appropriate pointer. Several agents may search different branches at the same time, provided their communication rules do not conflict on the same environmental objects.

### 6.2 Log and Result Representation

The log area $Z_L$ stores operation records before changes are written to the data area. A log record may contain the operation type, key, old value, new value, and a pointer to the affected row. In this first version of the architecture, logging is described at the conceptual level. A full recovery protocol is left for future work.

The result area $Z_R$ stores outputs of queries. For Boolean or counting queries, the final object $f$ can be used directly. Each matching tuple causes an agent to deposit one copy of $f$ in $Z_R$. The number of copies of $f$ then represents the result of a COUNT query. For projection queries, the result area may store encoded values copied from the selected rows.

## 7 Database Operations

This section outlines the realization of elementary database operations. The descriptions are intentionally abstract; they define the data-management behavior expected from agents rather than a complete list of all formal programs. The

index area of the environment is currently not exploited, since the agent programs operating on it have not yet been defined; we leave their design for future work, noting that the full potential of the proposed architecture remains to be realized.

### 7.1 INSERT

An INSERT operation adds a new tuple to the data area. A task object is first placed in the queue area $Z_Q$. An idle agent picks up the task and receives the state object *ins* and a value object encoding the tuple to be inserted.

The agent then moves to $Z_D$ and searches for a free row. A row is considered free if its marker cell contains no row marker or contains the deletion marker $d$. When a free row is found, the agent writes the marker $r$ and subsequently writes the encoded attribute values into the corresponding cells. After the row has been inserted, the agent deposits a re-indexing request $ri$ into the queue area. An index-maintenance agent later updates $Z_I$.

$$P_{Ins1} : \langle e \leftrightarrow ins \ ; \ e \rightarrow e \rangle \tag{1}$$

*pick up ins from queue* $Z_Q$

$$P_{Ins2} : \langle [e_{1,2}] \rightarrow\Rightarrow \ ; \ ins \rightarrow ins \rangle \tag{2}$$

*move right to next queue cell*

$$P_{Ins3} : \langle e \rightarrow ri_j \ ; \ ins \rightarrow ins \rangle \tag{3}$$

*rename carried* $e$ *to* $ri_j$*;* $j$ *is identifier of the agent*

$$P_{Ins4} : \langle ri_j \leftrightarrow v \ ; \ ins \rightarrow ins \rangle \tag{4}$$

*swap* $ri_j$ *task into cell, pick up* $v$

$$P_{Ins5} : \langle [e_{1,0}] \rightarrow\Leftarrow \ ; \ ins \rightarrow ins \rangle \tag{5}$$

*move left, not yet at boundary*

$$P_{Ins6} : \langle [B_{1,0}] \rightarrow\Downarrow \ ; \ ins \rightarrow ins_D \rangle \tag{6}$$

*left cell is* $B$*, column 0 reached, switch to* $ins_D$

$$P_{Ins7} : \langle [e_{2,1}] \rightarrow\Downarrow \ ; \ ins \rightarrow ins_D \rangle \tag{7}$$

*move down into* $Z_D$*, switch to row scan*

$$P_{Ins8} : \langle r \leftrightarrow ins_D \ ; \ v \rightarrow v \rangle \tag{8}$$

*column 0 holds $r$, row occupied*

$$P_{Ins9} : \langle ins_D \leftrightarrow r \ ; \ r \rightarrow ins_{check} \rangle \tag{9}$$

*put $r$ back, switch to $ins_{check}$*

$$P_{Ins10} : \langle [e_{2,1}] \rightarrow \Downarrow \ ; \ ins_{check} \rightarrow ins_D \rangle \tag{10}$$

*move down to next row, switch back to scan*

$$P_{Ins11} : \langle e/d \leftrightarrow ins_D \ ; \ v \rightarrow v \rangle \tag{11}$$

*column 0 holds $e$ or $d$, row free*

$$P_{Ins12} : \langle e \rightarrow e \ ; \ e \rightarrow r \rangle \tag{12}$$

*rename carried $e$ to $r$*

$$P_{Ins13} : \langle r \leftrightarrow e \ ; \ e \rightarrow ins_R \rangle \tag{13}$$

*deposit $r$ into column 0, switch to writing*

$$P_{Ins14} : \langle [e_{1,2}] \rightarrow \Rightarrow \ ; \ ins_R \rightarrow ins_R \rangle \tag{14}$$

*move right to target column*

$$P_{Ins15} : \langle e \leftrightarrow v \ ; \ ins_R \rightarrow ins_{done} \rangle \tag{15}$$

*write $v$ into cell, task complete*

$$P_{Ins16} : \langle [e_{0,1}] \rightarrow \Uparrow \ ; \ ins_{done} \rightarrow ins_{done} \rangle \tag{16}$$

*cell above is $e$, still in $Z_D$*

$$P_{Ins17} : \langle [B_{0,1}] \rightarrow \Uparrow \ ; \ ins_{done} \rightarrow ri \rangle \tag{17}$$

*cell above is $B$, in $Z_Q$, rename to $ri$*

$$P_{Ins18} : \langle [e_{1,2}] \rightarrow \Rightarrow \ ; \ ri \rightarrow ri \rangle \tag{18}$$

*move right to find $ri_j$ cell*

$$P_{Ins19} : \langle ri \leftrightarrow ri_j \ ; \ e \rightarrow e \rangle \tag{19}$$

*deposit $ri$ into queue, task done*

### 7.2 DELETE

A DELETE operation removes a tuple identified by a key. The agent picks up a task containing the state object *del* and the key object $k$. It first moves to the index area $Z_I$ and searches for the key. If the key is found, the agent obtains the pointer $p$ to the corresponding row in $Z_D$.

The agent then moves to the row and marks it as deleted by replacing the row marker or the stored values with the deletion object $d$. The deleted row is not physically removed from the environment. It remains available for future reuse by INSERT agents. After the deletion, the agent creates a re-indexing request so that the index no longer points to the deleted tuple.

The first programs $P_{Del1} - P_{Del5}$ load the task from $Z_Q$ and move the agent to column 0, are identical to the INSERT section, programs $P_{Ins1} - P_{Ins5}$. The following rules implement a sequential scan of $Z_D$, for this demonstration without using index structure. From $P_{Del5}$ onward the agent moves down into $Z_D$ and scans the key column for a matching key.

$$P_{Del6} : \langle [e_{2,1}] \rightarrow \Downarrow \; ; \; del \rightarrow del \rangle \tag{20}$$

*move down into $Z_D$*

$$P_{Del7} : \langle [e_{1,2}] \rightarrow \Rightarrow \; ; \; del \rightarrow del \rangle \tag{21}$$

*move right to key column*

$$P_{Del8} : \langle k_i \leftrightarrow del \; ; \; del \rightarrow del_{found} \rangle \tag{22}$$

*key matches $k_i$, agent carries $\{del_{found}, del\}$*

$$P_{Del9} : \langle del \leftrightarrow k_i \; ; \; del_{found} \rightarrow del_{found} \rangle \tag{23}$$

*restore $k_i$ to cell, agent carries $\{del_{found}, k_i\}$*

$$P_{Del10} : \langle [e_{1,0}] \rightarrow \Leftarrow \; ; \; del_{found} \rightarrow del_{found} \rangle \tag{24}$$

*move left back to column 0*

$$P_{Del11} : \langle r \rightarrow d \; ; \; del_{found} \rightarrow del_{done} \rangle \tag{25}$$

*mark row as deleted by writing d into column 0*

$$P_{Del12} : \langle [B_{0,1}] \rightarrow \Uparrow \; ; \; del_{done} \rightarrow e \rangle \tag{26}$$

*cell above is B, reached $Z_Q$, agent done with $\{e, e\}$*

$$P_{Del13} : \langle k_j \leftrightarrow del \ ; \ del \rightarrow del_{skip} \rangle \tag{27}$$

*key $k_j \neq k_i$, no match, agent carries $\{del_{skip}, del\}$*

$$P_{Del14} : \langle del \leftrightarrow k_j \ ; \ del_{skip} \rightarrow del_{skip} \rangle \tag{28}$$

*restore $k_j$ to cell*

$$P_{Del15} : \langle [e_{1,0}] \rightarrow\Leftarrow \ ; \ del_{skip} \rightarrow del_{skip} \rangle \tag{29}$$

*move left to column 0*

$$P_{Del16} : \langle [e_{2,1}] \rightarrow\Downarrow \ ; \ del_{skip} \rightarrow del \rangle \tag{30}$$

*move down to next row, switch back to scanning*

The last four programs $P_{Del17} - P_{Del20}$ are identical to INSERT programs $P_{Ins16} - P_{Ins19}$ and are responsible for adding a request $ri$ to the reindex queue.

### 7.3 UPDATE

An UPDATE operation changes one or more attribute values of an existing tuple. The agent receives the state object *upd*, a key object, and new encoded values. It uses the index to locate the row. Then it swaps the old value objects with the new value objects using communication rules.

The communication-rule swap is important because it provides a local atomic exchange primitive. If only non-key attributes are updated, the index remains valid. If the key attribute is changed, the agent deposits a re-indexing request into the queue area.

The UPDATE operations follow the same structure as DELETE. The first four rules are identical — the agent loads the task from $Z_Q$, picking up the key $k$ to locate and a new value $v$ to write. The scanning phase $PUpd6$ through $P_{Upd16}$ is also similar to $P_{Del6} - P_{Del16}$: the agent moves down $Z_D$ column by column searching for a matching key. The only difference lies in the match action: instead of writing $d$ into column 0 as DELETE does in $P_{11}$, the UPDATE agent moves right to the value column and swaps the old value $v$ for the new value $v$ using a communication rule, then returns to $Z_Q$ as in $P_{ins16} - P_{ins19}$.

### 7.4 SELECT

A SELECT operation reads data. There are two basic cases: key lookup and full scan.

In a key lookup, the agent receives the state object *sel* and a key object $k$. It searches the index area, obtains the pointer to the matching row, moves to the

data area, reads the encoded values, and deposits the result in $Z_R$. If the row is marked by $d$, no result is produced.

In a full scan, several agents may be created or activated simultaneously. Each agent receives a pointer to a different part of $Z_D$ and scans a subset of rows. When an agent finds a row satisfying the selection condition, it deposits one result object $f$ in $Z_R$ or copies the selected values into the result area. Thus, parallel query evaluation emerges from the number of scan tasks in the queue.

The full scan UPDATE operations follow the same structure as DELETE. The first four rules are identical — the agent loads the task from $Z_Q$, picking up the key k to locate and a new value $v$ to write. The scanning phase $P_{Ins6} - P_{Ins61}$ is also the same as $P_{Del6} - P_{Del16}$: the agent moves down $Z_Q$ column by column searching for a matching key. The only difference lies in the match action: instead of writing $d$ into column 0 as DELETE does in $P_{Ins11}$, the UPDATE agent moves right to the value column and swaps the old value $v$ for the new value $v$ using a communication rule, then returns to $Z_Q$.

### 7.5 Concurrency

The colony model provides a natural form of conflict avoidance: two communication rules cannot act on the same environmental object in the same computational step. Therefore, two agents cannot simultaneously modify the same cell. For operations involving several cells of the same row, a lock object $lk$ can be placed in the row marker cell. Other agents encountering $lk$ do not apply their programs to that row until the lock is removed.

This mechanism is weaker than a complete transaction isolation model in conventional database systems, but it is suitable as a first formal abstraction. It shows how concurrency control can be expressed in the native primitives of a 2D P colony rather than by importing external synchronization mechanisms.

### 7.6 Cost Measures

The model also provides natural cost measures. Agent moves correspond to spatial access cost. The number of visited cells approximates the cost of scans and index lookups. The number of result objects in $Z_R$ represents the cardinality of the result set. The number of deletion markers $d$ indicates fragmentation of the data area. These measures can later be used for query optimization and for comparing different index layouts.

## 8 Discussion

The proposed architecture should be understood as a formal and conceptual contribution. It does not claim that current biological computers can directly implement all mechanisms described in the paper. Rather, it identifies a missing architectural layer and shows that numerical 2D P colonies provide suitable primitives for modeling such a layer.

The main advantage of the approach is the combination of three features. First, the numerical representation allows arbitrary data to be encoded in a uniform way. Second, the two-dimensional environment provides a spatial storage substrate. Third, the agent-based computation provides inherent parallelism and local interaction. In principle, even a single agent could perform the required computational tasks by dynamically assuming different roles depending on the given task. However, the key advantage of the agent-based approach lies in its ability to exploit massive parallelism by distributing computation across multiple agents operating concurrently. The number of agents is not fixed and can be chosen according to the required computational performance.

Together, these features allow us to reinterpret database concepts such as tables, indexes, logs, and result buffers in the language of a bio-inspired computational model.

The architecture also has limitations. A complete relational database management system requires a query language, query optimization, schema management, integrity constraints, transaction management, recovery protocols, joins, aggregation, and possibly distributed storage. The present paper addresses only the basic storage and manipulation layer. This is intentional: the goal is to introduce the architecture and its fundamental operations, not to define a complete SQL engine inside a P colony.

Another limitation concerns physical realization. P colonies are formal bio-inspired models, not necessarily direct wet-lab implementations. Therefore, the phrase biological computer is used here in a broad sense including theoretical and bio-inspired computational architectures. The proposed model may serve as an intermediate formal layer between abstract biological computing and future physical implementations.

From an implementation perspective, the proposed architecture can be incorporated into the existing numerical 2D P colony by extending the representation of the environment with the logical areas described in Section 5 and by implementing the corresponding agent programs. Such an implementation would also provide a basis for systematic experimental evaluation. In the numerical 2D P colony model, the size of an individual stored value does not directly determine the cost of its manipulation: after encoding, a value is represented as a single numerical object handled by an agent. The principal access cost is therefore spatial and depends on the distance, expressed as the number of cells of the two-dimensional environment, that an agent must traverse to reach the required data, index, queue, or result position. Consequently, the spatial organization of the environment and the placement of related data become important performance factors. Scalability can be addressed using data fragmentation, a well-established approach in database management systems. A relation may be partitioned into several spatial fragments located in different regions of the environment, allowing agents to operate locally on the fragment relevant to a particular task and enabling independent fragments to be processed in parallel. Increasing the number of agents can provide additional parallelism for independent operations, while contention for the same environmental objects may limit the achievable

speedup. The expected performance therefore depends primarily on access distances, fragmentation and data placement, indexing strategy, and the number of concurrently operating agents. The cost measures introduced in Section 7.6, particularly the number of computational steps, agent movements, and visited cells, provide natural metrics for evaluating these effects. A systematic experimental analysis of fragmentation strategies, spatial layouts, and agent populations is left for future work.

Nevertheless, the architectural perspective is useful. It separates the problem of data management from the problem of computation itself. This separation is standard in classical computing but underdeveloped in biological computing. By formulating data management as an explicit research problem, the paper opens several directions for future work, including relational algebra over P colonies, graph-based data models, query optimization, transaction protocols, and simulation-based evaluation.

## 9 Conclusion

This paper proposed numerical 2D P colonies as a data management architecture for biological computers. The motivation comes from the observation that biological computing research focuses mainly on computation, while the organization and management of data inside biological computers remains insufficiently explored. We introduced a relational storage abstraction over the environment of a numerical 2D P colony and described how queue, data, index, log, and result areas can support elementary database operations.

The proposed architecture should be viewed as a first step toward systematic data management in biological computing. Future work will focus on formal definitions of the corresponding colony programs, implementation in an existing 2D P colony simulator, support for joins and relational algebra, experimental evaluation of index structures, and development of recovery and transaction mechanisms.

## Acknowledgments

This work was supported by the Silesian University in Opava under the Student Funding Plan, project SGS/9/2024.

## References

1. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science **266**(5187), 1021–1024 (1994). https://doi.org/10.1126/science.7973651
2. Cienciala, L., Ciencialová, L., Perdek, M.: 2d p colonies. In: Membrane Computing, Lecture Notes in Computer Science, vol. 7762, pp. 161–172. Springer (2012). https://doi.org/10.1007/978-3-642-36751-9_12
3. Cienciala, L., Ciencialová, L., Sosík, P.: P colonies with agent division. Information Sciences **589**, 162–169 (2022). https://doi.org/10.1016/j.ins.2021.12.094

4. Ciencialová, L., Cienciala, L., Sosík, P.: Generalized p colonies with passive environment. Theoretical Computer Science **724**, 61–68 (2018). https://doi.org/10.1016/j.tcs.2017.12.009
5. Kelemen, J., Kelemenová, A., Păun, G.: Preview of p colonies: A biochemically inspired computing model. In: Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems. pp. 82–86 (2004)
6. Langer, M., Valenta, D.: On evolving environment of 2d p colonies: Ant colony simulation. Journal of Membrane Computing **5**, 117–128 (2023). https://doi.org/10.1007/s41965-023-00123-y
7. Langer, M., Valenta, D., Patnaik, P.K.: On simulation of the 2d p colony with evolving environment (2026), manuscript
8. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
9. Păun, G.: Introduction to membrane computing. In: Applications of Membrane Computing, pp. 1–42. Springer (2006)
10. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, New York (2010)
11. Valenta, D., Langer, M.: On numerical 2d p colonies modelling the grey wolf optimization algorithm. Processes **9**(2), 330 (2021). https://doi.org/10.3390/pr9020330

# TENNCell: Transformation and Execution of Numerical Networks of Cells

Sergey Verlan[1], Junying Xiao[2], Gexiang Zhang[2], and Qiyu Liu[2]

[1] Univ. Paris Est Creteil, LACL, 94010, Creteil, France
verlan@u-pec.fr
[2] School of Automation, Chengdu University of Information Technology, Chengdu, China
3026262319@qq.com,{zhgxdylan,liuqy87}@126.com

**Abstract.** Numerical P systems and related variants have been widely used in applications such as robot-controller design, motion planning, testing, and FPGA-oriented implementation. Numerical networks of cells (NNC) generalize these models by making guarded numerical updates and value domains explicit. This paper presents TENNCell, a Python environment for executing and transforming numerical networks of cells. TENNCell uses YAML as a specification language and lowers each model into a shared internal representation of cells, variables, rules, constants, aliases, and imports. The same source can be simulated, embedded as a Python library, or transformed into Python, SystemVerilog, and Webots controller code. We describe the supported NNC fragment, the execution modes, the implementation architecture, and representative software, hardware, and robotics-oriented examples.

**Keywords:** Membrane computing · Numerical networks of cells · Numerical P systems · Simulation · Code generation · SystemVerilog · Robotics

## 1 Introduction

Membrane computing studies P systems, which are computational models inspired by the structure and functioning of living cells. They consist of compartments containing multisets of objects, and rules that evolve the contents of the compartments in a discrete-time manner. Numerous models have been proposed, we refer to [19,20,22,36] for a comprehensive overview. Numerical P systems [21] (NPSs) are a particular variant of P systems that consider numerical state variables rather than symbolic objects, as well as corresponding rules. Since their introduction, NPSs and related variants have moved beyond purely theoretical research: enzymatic numerical P systems (ENPS) have been used for mobile-robot controllers and robot localization [23,24,8], multi-robot and ROS-based controllers [11], RRT-based motion planning [26,25], and recent robot-controller testing methodologies [6,5]. On the implementation side, generalized numerical P systems have also been studied as a basis for FPGA-oriented designs [30,31].

Numerical networks of cells (NNCs) provide a general formal setting for this line of models. They abstract from particular NPS, ENPS, or GNPS syntax and emphasize cells equipped with numerical variables, guarded updates, and value domains that may vary from one instance to another [1,18]. This is important from the theoretical point of view, allowing more precise models, as well as from a practical point of view, because it makes possible to design software tools that are reusable across several variants of numerical P systems and that can be connected to software, hardware, and robotic simulation environments.

From a more general perspective, NNCs correspond to reaction-based view of piecewise-smooth [4], or switched systems where the update rule is given by a different function in each region of the state space. Typical examples of such systems include systems of conditional recurrences, systems of piecewise-defined difference equations and Extended Finite State Machines (EFSM) [17]. Such models are widely used in control theory and engineering [3,4], which makes the NNC model an interesting bridge between membrane computing and more general mathematical models used in engineering. More details on this relation can be found in [34].

Simulation tools are essential in membrane computing because they allow to inspect traces, debug rule interactions, validate modelling hypotheses, and understand the behavior of the systems. A number of simulators and frameworks have been developed for various classes of P systems, including numerical P systems. We will only mention here the most relevant ones as P-Lingua, MeCoSim, UPSimulator, PSimulator, and Snapse/WebSnapse [15,27,16,28,10,9]. For NPS there are two relevant simulators SNUPS and PeP/GPUPeP [2,7,12,29]. We discuss these tools in more detail in Section 2.

One common point of the above tools is that they are primarily focused on simulation. This is useful when designing and testing a model, but it does not directly integrate into workflows where the model is only a small part of a larger software, hardware, or robotic system. For example, a robot controller may be designed as an ENPS model, but it must then be integrated into a Webots controller or a larger microcontroller-based design.

This paper presents TENNCell, a simulator and code generation environment whose name abbreviates *Transformation and Execution of Numerical Networks of Cells*. TENNCell accepts YAML-based DSL specifications describing an NNC model, in particular cells, variables and rules. It also adds syntactic sugar like constants, aliases, imports, and control structures. The input layer normalizes and lowers the YAML syntax into a shared in-memory NNC model. This model can then be executed directly by a Python runtime simulator or transformed into a standalone Python source code, SystemVerilog RTL-oriented modules, or Webots Python controllers. Thus, same description can be used for fast simulation, integration into Python-based experiments, and generation of code for hardware and robotic applications. This common-source workflow reduces the risk that a simulation model and an implementation model silently diverge.

The paper is organized as follows. Section 2 reviews existing P systems and NPS simulation tools. Section 3 introduces the supported NNC model and step

semantics. Section 4 presents the implementation architecture and execution modes as well as the code generation targets. Section 5 describes the input YAML DSL language and provides some representative examples. Finally, Section 6 gives the conclusions and discusses the further work.

## 2 Related Work

This section discusses existing simulation tools in order to give the context in which TENNCell should be placed. We first briefly discuss existing P systems simulation tools and then provide more details on numerical P systems simulators. Table 1 gives a short summary of the main existing simulation tools in the P systems area.

### 2.1 Simulation Tools for P Systems

Simulation has always played an important role in the practical use of P system models. Even when the final goal is a mathematical result or a physical implementation, a simulator is often the first tool used to inspect traces, understand the effect of rule priorities or parallelism, and validate a modelling hypothesis. For this reason, the membrane computing community has developed several simulation environments.

P-Lingua and PLinguaCore provide a language and a comprehensive framework for many cell-like P systems [15]. They are important for the general P system software ecosystem, but their target is not numerical P systems. MeCoSim builds configurable simulation interfaces over P system models and has been used to support virtual experiments in membrane computing [27]. UPSimulator is another general P system simulator supporting several P system classes [16]. PSimulator is a command-line tool for P system simulation, based on the network of cells and formal framework formalization of P systems [28,14,33]. Snapse and WebSnapse are visual tools for spiking neural P systems [10,9]. The above tools show the importance of simulation in the area, but their main targets are ordinary and not numerical P systems.

### 2.2 Numerical P system Simulators

For numerical P systems there are two simulators: SNUPS and PeP/GPUPep. SNUPS was the first freely available software tool for modelling and simulation of numerical P systems [2,7]. It features a graphical interface and an XML input/output format. However, at the present moment the project is not maintained and the source code is not publicly available.

PeP is an open-source Python simulator for numerical P systems and enzymatic numerical P systems [12,13]. It is a command-line simulator that integrates easily with Python-based experiments and workflows. It additionally supports trigonometric and generic numerical functions. It was used in numerous projects, including robot controllers and motion planning. GPUPeP extends this line with

a GPU-oriented parallel simulator still keeping same Python-based interface [29]. However, their emphasis is still simulation and acceleration of NPS/ENPS computations, with limited integration possibilities.

**Table 1.** Selected simulators and frameworks for P system models.

| Simulator | Target model(s) | Particularities |
|---|---|---|
| P-Lingua, PLinguaCore [15] | Cell-like P systems and related models | Programming language and software framework for defining P system models. |
| MeCoSim [27] | P system models, especially biological | General-purpose environment for running virtual experiments over P system models. |
| UPSimulator [16] | Cell-like, tissue-like, and neural-like P systems | General P system simulator supporting several major membrane-computing families within a unified simulation tool. |
| PSimulator [28] | Network-of-cells models with bisimulations to several P system variants | Command-line simulator for P systems given in terms of a network-of-cells descriptions; can optionally generate a state graph for further analysis. |
| Snapse, WebSnapse [10,9] | Spiking neural P systems | Visual and web-based tools for creating, modifying, simulating, and inspecting computations of SN P systems. |
| SNUPS [2,7] | Numerical P systems | Visual simulation tool with XML input/output format. |
| PeP [12] | Numerical P systems and enzymatic numerical P systems | Open-source Python simulator with support for NPS/ENPS input files, stepwise execution, bounded simulation, CSV output, enzymes, trigonometric functions, and generic numerical functions. |
| GPUPeP [29] | Enzymatic numerical P systems | GPU-oriented continuation of the PeP line; uses a Python-based interface and CUDA-based parallel execution for ENPS simulations. |

Hence, the main novelty of TENNCell is the simulation of more complex guarded rules as well as the transformation of the same NNC source model into Python, SystemVerilog, and Webots artefacts.

## 3 Formal Model and Step Semantics

In this section we briefly present the formal model of numerical networks of cells and its step semantics. For more details, we refer to [1,35].

### 3.1 NNC Model

The NNC model is defined over an arbitrary monoid. However, in this paper we consider the base domain equal to $\mathbb{R}$. An NNC model contains a finite set of cells $C_1, \ldots, C_n$, a finite set of variables $V_1, \ldots, V_m$, a placement function assigning each variable to a cell (hence each variable belongs to some cell), and distinguished sets of input and output variables $I, O \subseteq V$. From a technical point of view, the variables are real-valued discrete-time series. Input variables are supplied by the environment at each time step, output variables expose values to the environment at each step.

A configuration $C(t)$ at time $t$ consists of values of all variables at that time, grouped by their cell: $C(t) = (C_1(t), \ldots, C_n(t))$, where $C_i(t) = (v_{i1}(t), \ldots, v_{ik}(t))$ for some variables $v_{ij}$. The initial configuration is given by an initial valuation of all variables.

Rules are specified as:

$$r : g_r(x_1, \ldots, x_s) \mid f_r(y_1, \ldots, y_m) \to x,$$

where $g_r$ is a Boolean guard, $f_r$ is a numerical expression, $x$ is the target (consumer) variable, and $x_1, \ldots, x_s, y_1, \ldots, y_m$ are variables that may appear in the guard and in the expression. The variables appearing in the rule induce a structure between the cells, which can be read as a hypergraph, however graph and tree structures are of a particular interest.

### 3.2 Step Semantics

We say that a rule is applicable in configuration $C(t)$ if its guard evaluates to true over $C(t)$, e.g. $g(x_1(t), \ldots, x_s(t)) = \mathit{true}$.

Let $R$ be the set of all rules in the system. We denote by $\mathit{Appl}(R, C, t)$ the set of applicable rules in configuration $C(t)$, i.e. $\mathit{Appl}(R, C, t) = \{r \in R \mid r \text{ is applicable in } C(t)\}$. We also denote by $\mathit{Appl}_x(R, C, t)$ the set of applicable rules that target variable $x$ in configuration $C(t)$.

When $\mathit{Appl}_x(R, C, t)$ has more than one rule, there might be several parallel application strategies, called derivation modes. In what follows, we consider only the *all-parallel* derivation mode where all rules are applied in parallel, which is the most common one in applications. In this mode, each variable $x$ is updated by the simultaneous contribution of all applicable rules that target it:

$$x(t+1) = 0 + \sum_{r \in \mathit{Appl}_x(R,C,t)} f_r(y_{r1}(t), \ldots, y_{rm_r}(t)), \tag{1}$$

We observe that if no rule targets $x$ at step $t$, then $x(t+1) = 0$. In other words, variables are reset to zero at each step before being eventually updated by some rule(s). This is the default reset discipline in NNCs. However, in many NPS based models, variables are reset to zero only if they are effectively used in some production function at the current step:

$$x(t+1) = x'(t) + \sum_{r \in \mathit{Appl}_x(R,C,t)} f_r(y_{r1}(t), \ldots, y_{rm_r}(t)), \tag{2}$$

where $x'(t) = 0$ if there exists $r \in \mathit{Appl}(R, C, t)$, such that $x$ appears in $f_r$, and $x'(t) = x(t)$ otherwise.

In TENNCell, the user can select either of the above reset disciplines by enabling or disabling the `zero_reset_mode` option in module parameters. When its value is true, the first discipline is used, and when its value is false, the second discipline is used. This parameter is optional and its default value is false. This means that by default, NPS style reset discipline is used.

It is not difficult to see that NPS, ENPS, GNPS and other related models can be directly described using such rules. We refer to the paper [1] for more details on this relation.

### 3.3 Module Composition

TENNCell adds an explicit module layer to the core NNC model. A module may import other modules and connect local expressions to their inputs and outputs. After import resolution, a composed specification can be viewed as a single NNC model with a standard all-parallel synchronous step semantics.

## 4 Implementation Architecture

TENNCell is implemented as a Python package and distributed via PyPI. The main implementation repository is available at GitHub [32]. The choice of Python for the implementation is driven by pragmatic integration reasons. Python is widely used as a glue language in scientific computing, testing, robotics, and hardware-design workflows, and many external frameworks provide Python bindings or controller APIs. A Python implementation therefore makes the simulator usable directly in larger experiments and makes it natural to generate Python artifacts for fast backend-equivalence tests before moving to Webots or SystemVerilog.

From the architecture perspective, TENNCell is organized around a shared intermediate model. Figure 1 shows the main pipeline. The domain specification language (DSL) is YAML-based and described in Section 5. YAML files are loaded, normalized, and lowered into an in-memory model containing cells, variables, rules, constants, aliases, imports, and module metadata. The runtime simulator and all export backends consume this same model.

### 4.1 Library Interface

The PyPI distribution is named `tenncell`, while the main importable Python package is `nnc`. The principal public objects are cells, rules, and a system object loaded from YAML. The main system class is `NncSystem`.

TENNCell can therefore be used not only as a command-line program, but also as an embedded library in larger Python projects. A control application, experiment driver, test suite, or generated-code validation script can load an NNC model once and call its step function repeatedly, supplying input valuations programmatically and consuming the returned outputs. Table 2 summarizes the main public interface of the runtime package. The API is designed to be minimal and focused on the core NNC model, while the code generation backends are accessed via separate command-line tools.

Listing 1.1 shows the intended Python-level API. The import path uses `nnc`, whereas installation uses the PyPI distribution name `tenncell`.

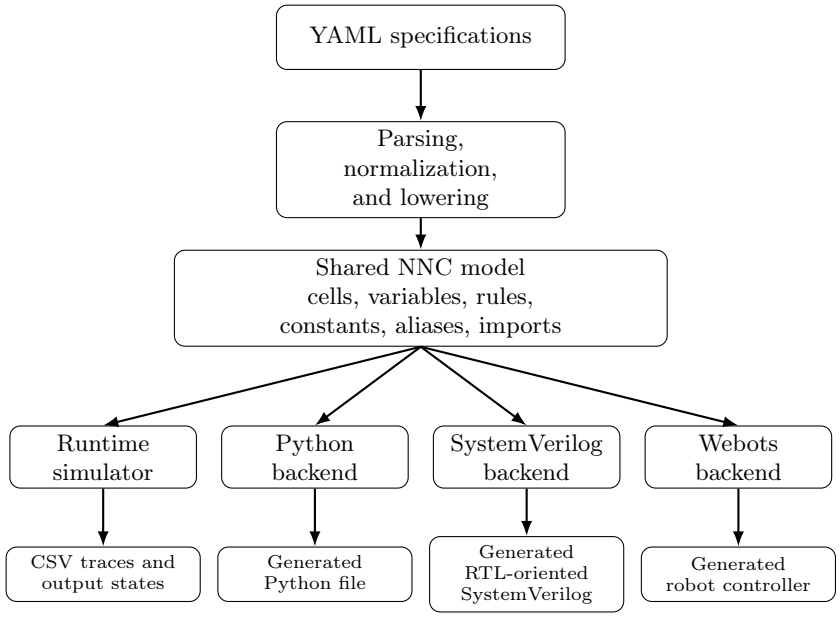



**Fig. 1.** TENNCell architecture. A single lowered NNC model is shared by the simulator and by the Python, SystemVerilog, and Webots code generators.

**Listing 1.1.** Minimal embedded use of the TENNCell Python API.

```
from nnc import NncSystem

system = NncSystem.from_yaml("examples/simple/example1.yaml")
outputs = system.step({"x": 1.0})
state = system.get_variables()
print(outputs, state)
```

### 4.2 Command-Line Execution Modes

The distribution provides two command-line tools. The command `nnc-sim` executes YAML-described NNC models and supports two execution modes, shown in Figure 2. In *IO mode*, the simulator acts as a CSV transducer. It reads an input CSV stream whose columns are NNC input variables, treats each row as the input valuation for one step, executes one step per row, and writes a CSV stream containing the declared output variables. This mode is useful for replaying experiments, testing controllers on recorded sensor traces, and comparing simulator outputs with generated-code outputs.

In *compute mode*, the model is run for a bounded number of steps without a CSV input stream. This mode is intended for models with no external input variables. The command reports the initial output state at step 0 and then the output state after each step. With the CSV option, the same bounded run is written as a CSV file with a `step` column. Compute mode is convenient for

**Table 2.** Main public interface of the `nnc` package.

| Interface item | Purpose |
|---|---|
| `NncSystem.from_yaml(path, import_paths=None)` | Load an NNC model from YAML, resolve imports, and validate references. |
| `system.step(inputs=None)` | Advance the model by one step and return the declared outputs. |
| `system.get_variables()` | Return the current internal valuation of all model variables. |
| `system.get_variable(name)` | Retrieve a single variable object by name. |
| `Cell`, `Rule` | Programmatic building blocks for constructing or inspecting models. |

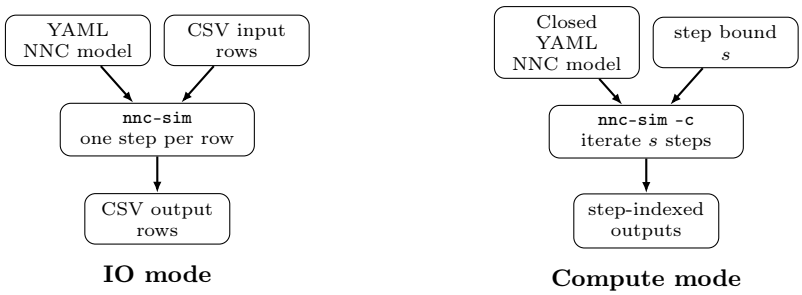



**Fig. 2.** The two execution modes of `nnc-sim`. IO mode consumes one input row per step and produces one output row per step; compute mode runs a closed NNC model for a bounded number of steps.

autonomous dynamical examples such as classical NPS/ENPS examples, oscillators, population models, counters, or internally clocked controllers.

The second command-line tool provided by the package is `nnc-gen`. It accepts the same YAML-based NNC specifications but instead of executing them, it generates code for a selected backend. The user can select the Python, SystemVerilog, or Webots backend by passing a command-line option. The generated code is written to a file or to standard output.

### 4.3 Code Generation Backends

The export layer transforms the shared model into executable or deployable artefacts. The current implementation contains three backends.

**Python Backend** The Python backend emits a single Python source file for a selected NNC model. For composed systems having several imported modules, the generated file contains a flattened representation of all necessary components. The generated class exposes a step method, making the code suitable for inspection, testing, and integration into larger Python workflows. It also

contains a simple command-line interface for running the generated code with CSV input and output, similar to the `nnc-sim` IO mode. The generated code is self-contained and does not rely on the TENNCell runtime, making it suitable for distribution and use in environments where the full simulator may not be available.

This backend has two roles. First, it is a convenient deployment target for users who want to distribute an executable model rather than a YAML specification plus simulator. Second, it provides a useful reference for checking that the transformation layer preserves the behavior of the runtime simulator.

**SystemVerilog Backend** SystemVerilog is a hardware description language widely used for designing and verifying digital circuits. The SystemVerilog backend generates RTL-oriented modules that implement the NNC semantics. Each input file is translated into a module containing registers for the variables and combinational logic for the rules. The generated code is structured to reflect the NNC step semantics, taking into account the reset discipline. For performance and clarity reasons, it is recommended to use the `zero_reset_mode` when targeting SystemVerilog, as this allows the generated code to avoid unnecessary state retention and simplifies the logic for variable updates.

The generated modules use explicit boundary information from the `verilog` YAML section, including clock and reset signals, top-level ports, numeric encodings, and optional wiring to external RTL modules. The generated structure separates combinational next-state computation from registered state updates, following the usual distinction between `always_comb` and `always_ff` blocks. The real values are encoded using fixed-point representations and automatc conversion helpers are generated to convert between the internal binary representation and the external connections.

The backend is motivated by FPGA-oriented numerical P system applications. It allows a TENNCell specification to serve as a higher-level rule-based description while still producing RTL-oriented code that can be inspected and integrated into a hardware design flow.

**Webots Backend** The Webots backend generates Python controller glue for the Webots robotics simulator. A `webots` YAML section binds NNC input variables to device read methods and NNC output variables to device write methods. The generated controller repeatedly reads sensor values, performs one NNC step, and writes actuator values. This allows a single TENNCell specification to serve as a high-level description of a robot controller, while the generated code can be directly deployed inside an established robotics simulation environment.

### 4.4 Tests

The current TENNCell repository includes automated tests at several levels. Unit tests exercise expression parsing, YAML loading, model validation, and

backend components. Functional tests compare simulator behavior on representative models. CLI tests check the command-line interface, and transformer tests compare generated Python or SystemVerilog artifacts against expected outputs.

## 5 The YAML DSL Specification Language

The input language for TENNCell is a YAML-based DSL. The choice of YAML is pragmatic: it is human-readable, easy to edit, and well suited to structured metadata. Each file describes a module, which may import other modules and connect to their inputs and outputs. The module contains declarations for cells, variables, rules, constants, aliases, and optional backend-specific metadata. The `module` section contains the module name, an optional description and an optional `zero_reset_mode` flag. All variables should be declared in some contents section of some cell, and all rules should be declared globally in the rules section. Each cell may contain optional input and output lists, which are used to declare the variables that serve as inputs and outputs of the system.

Listing 1.2 shows a system computing the mean, max and min values for input values $a, b, c$ at each step.

**Listing 1.2.** Min/max/mean computation.

```
module:
  name: mmm
  description: "Mean/Max/Min example"
cells:
  - id: 1
    contents:
      - a=0, b=0, c=0
      - min=0, max=0, mean=0
    input: [a, b, c]
    output: [min, max, mean]
rules:

  - a >= b && a >= c | a -> max
  - b > a && b >= c | b -> max
  - c > a && c > b | c -> max
  - a <= b && a <= c | a -> min
  - b < a && b <= c | b -> min
  - c < a && c < b | c -> min

  - (a + b + c) / 3 -> mean
  - 0 * mean -> mean
  - 0 * min -> min
  - 0 * max -> max
```

The final three rules in Listing 1.2 are resetting the output variables to zero before each step. Because by default TENNCell uses the NPS-style reset discipline, variables are only reset to zero if they are effectively used in some production function at the current step. In this example, the output variables `min`, `max`, and `mean` are only updated by one rule each, and they are not used in any other rule. Therefore, without the final three rules, these variables would sum up to their previous values. By adding the final three rules that multiply the variable by zero, we ensure that they are reset to zero. This is a common pattern in NPS-style models to ensure that variables do not carry over values

from previous steps when they are not explicitly updated. The alternative is to enable `zero_reset_mode`, where non-input variables are cleared automatically at every step, then the last three rules are not needed.

TENNCell also supports conditional YAML blocks. Such blocks are syntactic sugar: the loader lowers them into ordinary guarded rules. Listing 1.3 shows the style used in a Lotka–Volterra example, where each update is guarded to avoid negative populations.

**Listing 1.3.** Conditional rules in a Lotka–Volterra-style model.

```
module:
  name: lotka_volterra
constants:
  dt: 0.01
  a: 1.0
  b: 0.1
  d: 0.075
  g: 1.5
rules:
  - t + dt -> t
  - if: x + dt * (a*x - b*x*y) > 0
    then: x + dt * (a*x - b*x*y) -> x
    else: 0*x -> x
  - if: y + dt * (d*x*y - g*y) > 0
    then: y + dt * (d*x*y - g*y) -> y
    else: 0*y -> y
```

Finite-state-machine blocks are another source-level convenience. An FSM block declares a state variable, an initial state, and state-specific rules. The loader lowers this notation into ordinary NNC variables, constants, and guarded rules. This is useful when a numerical P system model includes controller-like discrete modes in addition to numerical dynamics.

**Listing 1.4.** FSM example: Counter with explicit state transitions.

```
module:
  name: fsm_counter_demo
constants:
  MAX_COUNT: 3
cells:
  - id: 1
    contents:
      - ctrl_state = 0, counter = 0, start = 0, done = 0
    input: [start]
    output: [counter, done]

fsm:
  - name: ctrl
    variable: ctrl_state
    initial: IDLE
    states:
      - IDLE:
          rules:
            - if: start > 0
              then:
                - RUN -> ctrl_state
                - counter * 0 -> counter
                - done * 0 -> done
              else: done * 0 -> done
      - RUN:
          rules:
            - if: counter >= MAX_COUNT
              then:
```

```
            - DONE -> ctrl_state
            - 1 -> done
          else: counter + 1 -> counter
    - DONE:
        rules:
          - done -> done
```

Backend-specific metadata is placed in dedicated sections. For example, a Webots section maps NNC variables to Webots devices and methods, while a Verilog section describes clock and reset signals, ports, real-number encodings, and external RTL modules. These sections do not change the core simulator semantics, rather, they provide the additional information needed by the corresponding code generator.

Listing 1.5 shows the central part of an e-puck controller. The NNC rules compute a sensor error from left and right proximity sensors, then adjust wheel speeds proportionally. The Webots metadata binds NNC variables to Webots devices and methods.

**Listing 1.5.** A Webots-oriented e-puck controller fragment.

```
module:
  name: e_puck_controller
constants:
  BASE_SPEED: 2.0
  KP: 0.5
rules:
  - (right_sensor - left_sensor) * KP -> sensor_error
  - BASE_SPEED - sensor_error -> left_speed
  - BASE_SPEED + sensor_error -> right_speed
webots:
  controller_name: e_puck_pid_controller
  timestep: 64
  bindings:
    left_sensor:
      device: ps0
      read_method: getValue
    right_sensor:
      device: ps7
      read_method: getValue
    left_speed:
      device: left wheel motor
      write_method: setVelocity
    right_speed:
      device: right wheel motor
      write_method: setVelocity
```

The FPGA examples in the project folder show how an NNC model can be complemented with hardware-specific metadata. Typical examples include LED blinking, UART-to-LED communication, SPI/GPIO bridges, and sensor controllers. In these cases, the NNC rules describe the numerical or control behavior, while the Verilog section supplies the information required to generate a hardware boundary.

Listing 1.6 shows a simple LED blink example for the Tang Nano 20K FPGA board. The NNC rules implement a counter that toggles the LED state after a certain delay. The Verilog section specifies the clock and reset signals, the LED output port, and the fixed-point encoding for real values.

**Listing 1.6.** An FPGA Tang Nano 20K LED blink example with active-low output

```yaml
module:
  name: fpga_blink
  zero_reset_mode: true
constants:
  BLINK_DELAY: 27000000
cells:
  - id: 1
    contents:
      - counter = 0
      - led = 0
    output: [led]
rules:
  - if: counter < BLINK_DELAY
    then:
      # Increment the counter
      - counter + 1 -> counter
      # Keep the led state (otherwise it resets to zero)
      - led -> led
    else:
      # When the delay expires, toggle the LED.
      # The counter resets to zero by itself (because of the zero_reset_mode)
      - led == 0 | 1 -> led
      - led > 0 | 0 -> led
verilog:
  real_encoding:
    kind: fixed_point
    signed: false
    width: 48
    frac_bits: 10
  clock: clk
  reset: rst
  ports:
    - name: led
      direction: output
      kind: logic
      width: 1
      signed: false
```

## 6 Conclusions

This paper presented TENNCell, a YAML-based environment for the transformation and execution of numerical networks of cells. TENNCell provides a declarative language for cells, variables, constants, aliases, imports, and guarded production rules. It executes models step by step, supports modular composition, and transforms the same internal NNC model into Python, SystemVerilog, and Webots controller code. The framework therefore provides a bridge from numerical networks of cells and related numerical P system specifications to executable artefacts for simulation, software integration, hardware-oriented design, and robotic simulation.

TENNCell is currently a practical simulator and code generator, not a proof assistant. The equivalence between the runtime simulator and each generated backend is tested, but not yet formally verified. The numerical domain is implemented through Python-level floating-point values and backend-specific encodings, so users must pay attention to numerical precision when moving from simulation to hardware.

Future work includes formalizing backend equivalence, adding richer static analyses, extending the set of supported numerical types, in particular to arrays, adding more control constructs, improving hardware-generation reports, adding benchmark suites, and integrating more deeply with robotics and FPGA toolchains. Another useful direction is the development of visual editors or trace viewers for NNC models, which would make the framework more accessible to users who are not comfortable editing YAML directly.

## Acknowledgments

This work was supported by the Sichuan Science and Technology Program (2025YFHZ0103 and 2025HJRC0022).

## References

1. Alhazov, A., Freund, R., Ivanov, S., Verlan, S.: Numerical networks of cells. Theoretical Computer Science **958**, 113873 (2023). https://doi.org/10.1016/j.tcs.2023.113873
2. Arsene, O., Buiu, C., Popescu, N.: SNUPS—a simulator for numerical membrane computing. International Journal of Innovative Computing, Information and Control **7**(6), 3509–3522 (2011)
3. Åström, K.J., Wittenmark, B.: Computer-Controlled Systems: Theory and Design. Dover Publications, New York, 3rd revised dover edition edn. (2011)
4. di Bernardo, M., Budd, C.J., Champneys, A.R., Kowalczyk, P.: Piecewise-smooth Dynamical Systems: Theory and Applications, Applied Mathematical Sciences, vol. 163. Springer, London (2008). https://doi.org/10.1007/978-1-84628-708-4
5. Bobe, R.T., Gheorghe, M., Ipate, F., Niculescu, I.M.: Test-driven simulation of robots controlled by enzymatic numerical P systems models. In: Simulation Tools and Techniques. 15th EAI International Conference, SIMUtools 2023. pp. 56–69. Springer (2024). https://doi.org/10.1007/978-3-031-57523-5_5
6. Bobe, R.T., Ipate, F., Niculescu, I.M.: Modelling and search-based testing of robot controllers using enzymatic numerical P systems. In: Proceedings 7th Symposium on Working Formal Methods (FROM 2023). Electronic Proceedings in Theoretical Computer Science, vol. 389, pp. 1–10 (2023). https://doi.org/10.4204/EPTCS.389.1
7. Buiu, C., Arsene, O., Cipu, C., Patrascu, M.: A software tool for modeling and simulation of numerical P systems. BioSystems **103**(3), 442–447 (2011). https://doi.org/10.1016/j.biosystems.2010.11.013
8. Buiu, C., Vasile, C.I., Arsene, O.: Development of membrane controllers for mobile robots. Information Sciences **187**, 33–51 (2012). https://doi.org/10.1016/j.ins.2011.10.007
9. Dupaya, A.G.S., Galano, A.C.A.P., Cabarle, F.G.C., de la Cruz, R.T., Ballesteros, K.J., Lazo, P.P.L.: A web-based visual simulator for spiking neural P systems. Journal of Membrane Computing **4**, 21–40 (2022). https://doi.org/10.1007/s41965-022-00092-8
10. Fernandez, A.D.C., Fresco, R.M., Cabarle, F.G.C., de la Cruz, R.T.A., Macababayao, I.C.H., Ballesteros, K.J., Adorna, H.N.: Snapse: A visual tool for spiking neural P systems. Processes **9**(1), 72 (2021). https://doi.org/10.3390/pr9010072

11. Florea, A.G., Buiu, C.: Modelling multi-robot interactions using a generic controller based on numerical P systems and ROS. In: 2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI). pp. 1–6. IEEE (2017). https://doi.org/10.1109/ECAI.2017.8166411
12. Florea, A.G., Buiu, C.: PeP: An open-source software simulator of numerical P systems and numerical P systems with enzymes. https://github.com/andrei91ro/pep (2017)
13. Florea, A.G., Buiu, C.: PeP (enzymatic) numerical P system simulator documentation. https://andrei91ro.github.io/pep/ (2017)
14. Freund, R., Verlan, S.: A formal framework for static (tissue) P systems. In: Eleftherakis, G., Kefalas, P., Paun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 4860, pp. 271–284. Springer (2007). https://doi.org/10.1007/978-3-540-77312-2_17
15. García-Quismondo, M., Gutiérrez-Escudero, R., Martínez-del Amor, M.A., Orejuela-Pinedo, E., Pérez-Hurtado, I.: P-lingua 2.0: A software framework for cell-like P systems. International Journal of Computers, Communications and Control **4**(3), 234–243 (2009). https://doi.org/10.15837/ijccc.2009.3.2431
16. Guo, P., Quan, C., Ye, L.: UPSimulator: A general P system simulator. Knowledge-Based Systems **170**, 20–25 (2019). https://doi.org/10.1016/j.knosys.2019.01.013
17. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines-a survey. Proceedings of the IEEE **84**(8), 1090–1123 (1996). https://doi.org/10.1109/5.533956
18. Liu, J., Wang, L., Zhang, G., Verlan, S., Zhu, M.: Universal enzymatic numerical P systems with small number of enzymatic rules. Theoretical Computer Science **1004**, 114630 (2024). https://doi.org/10.1016/j.tcs.2024.114630
19. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
20. Păun, G.: Membrane Computing: An Introduction. Natural Computing Series, Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/978-3-642-56196-2
21. Păun, G., Păun, R.A.: Membrane computing and economics: Numerical P systems. Fundamenta Informaticae **73**(1–2), 213–227 (2006). https://doi.org/10.3233/FUN-2006-731-220
22. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford (2010). https://doi.org/10.1093/oso/9780199556670.001.0001
23. Pavel, A.B., Buiu, C.: Using enzymatic numerical P systems for modeling mobile robot controllers. Natural Computing **11**(3), 387–393 (2012). https://doi.org/10.1007/s11047-011-9286-5
24. Pavel, A.B., Vasile, C.I., Dumitrache, I.: Robot localization implemented with enzymatic numerical P systems. In: Biomimetic and Biohybrid Systems. Lecture Notes in Computer Science, vol. 7375, pp. 204–215. Springer (2012). https://doi.org/10.1007/978-3-642-31525-1_18
25. Pérez-Hurtado, I., Martínez-del Amor, M.Á., Zhang, G., Neri, F., Pérez-Jiménez, M.J.: A membrane parallel rapidly-exploring random tree algorithm for robotic motion planning. Integrated Computer-Aided Engineering **27**(2), 121–138 (2020). https://doi.org/10.3233/ICA-190616
26. Pérez-Hurtado, I., Pérez-Jiménez, M.J., Zhang, G., Orellana-Martín, D.: Simulation of rapidly-exploring random trees in membrane computing with P-lingua and

automatic programming. International Journal of Computers, Communications and Control **13**(6), 1007–1031 (2018). https://doi.org/10.15837/ijccc.2018.6.3370
27. Pérez-Hurtado, I., Valencia-Cabrera, L., Pérez-Jiménez, M.J., Colomer, M.A., Riscos-Núñez, A.: MeCoSim: A general purpose software tool for simulating biological phenomena by means of P systems. In: 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA). pp. 637–643. IEEE (2010). https://doi.org/10.1109/BICTA.2010.5645199
28. PSimulator: command line simulator for P systems, https://github.com/sverlan/psimulator
29. Raghavan, S., Rai, S.S., Rohit, M.P., Chandrasekaran, K.: GPUPeP: Parallel enzymatic numerical P system simulator with a python-based interface. BioSystems **196**, 104186 (2020). https://doi.org/10.1016/j.biosystems.2020.104186
30. Shang, Z., Verlan, S., Zhang, G., Rong, H.: FPGA implementation of numerical P systems. International Journal of Unconventional Computing **16**(2–3), 279–302 (2021)
31. Shang, Z., Wei, Z., Verlan, S., Li, J., He, Z.: An FPGA architecture for the RRT algorithm based on membrane computing. Electronics **12**(12), 2741 (2023). https://doi.org/10.3390/electronics12122741
32. Tenncell github repository, https://github.com/sverlan/tenncell
33. Verlan, S., Zhang, G.: A tutorial on the formal framework for spiking neural P systems. Natural Computing **22**(1), 181–194 (2023). https://doi.org/10.1007/s11047-022-09896-0
34. Verlan, S., Zhang, G., Liu, Q., Dong, J.: Numerical P systems in a unifying view of discrete dynamical systems, automata and physical computing. Submitted (2026)
35. Zhang, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Verlan, S., Konur, S., Hinze, T., Gheorghe, M.: Membrane Computing Models: Implementations. Springer Singapore (2021). https://doi.org/10.1007/978-981-16-1566-5
36. Zhang, G., Verlan, S., Wu, T., Cabarle, F.G.C., Xue, J., Orellana-Martín, D., Dong, J., Valencia-Cabrera, L., Pérez-Jiménez, M.J.: Spiking Neural P Systems: Theory, Applications and Implementations. Springer Nature Singapore (2024). https://doi.org/10.1007/978-981-97-9282-5

# Author Index


Abucay, Jacky John D. 13
Adorna, Henry N. 309
Alhazov, Artiom 43

Balzerani, Matteo 67
Battyányi, Péter 83
Bobe, Radu Traian 99
Buño, Kelvin C. 13

Cabarle, Francis George C. 13, 309, 393
Ceterchi, Rodica 207
Cienciala, Luděk 127
Ciencialová, Lucie 127
Constantin, Doru 145
Crespo, Jerónimo 163

Dietrich, Lucas 179

Erba, Sandro 67

Förster, Benjamin 179
Franco, Giuditta 67
Freund, Rudolf 43

Gheorghe, Marian 145, 207, 225
Goñi-Moreno, Ángel 3

Hinze, Thomas 179

Iacob, Ștefan 145, 225
Ipate, Florentin 99, 145, 225
Ivanov, Sergiu 43

Konur, Savas 99

Langendörfer, Peter 179
Langer, Miroslav 447
Lázár, Katalin Anna 255
Lefticaru, Raluca 5
Leporati, Alberto 283
Linares-Barranco, Alejandro 361
Lipat, Job J. 309
Liu, Qiyu 465
Llanes, Rodrigo 425

Martínez-del-Amor, Miguel Ángel 309
Messner, Annemarie-Beatrix 331

Nagar, Atulya 207
Niculescu, Ionuț Mihai 145, 225
Nishida, Taishin Y. 407

Odhuno, Shane I. 13
Orellana-Martín, David 163, 309, 361, 375, 393

Paul, Prithwineel 7
Păun, Andrei 331
Pérez-Jiménez, Mario J. 375

Rajhona, Leóna Anasztázia 255
Ramanujan, Ajeesh 207
Ramírez-de-Arellano-Marrero, Antonio 393
Real, Pedro 345
Riscos-Núñez, Agustín 361

Sempere, José M. 425
Subramanian, K.G. 207

Tudose, Cristina 145

Valencia-Cabrera, Luis 375
Valenta, Daniel 447
Verlan, Sergey 43, 465

Wang, Tao 9

Xiao, Junying 465

Zandron, Claudio 67
Zhang, Gexiang 465