Formal Verification of P Systems

Ciprian Dragomir

Department of Computer Science

University of Sheffield

A thesis submitted for the degree of

Doctor of Philosophy

April, 2016

Abstract

Membrane systems, also known as P systems, constitute an innovative computational paradigm inspired by the structure and dynamics of the living cell. A P system consists of a hierarchical arrangement of compartments and a finite set of multiset rewriting and communication rules, which operate in a maximally parallel manner. The *organic* vision of concurrent dynamics captured by membrane systems stands in antithesis with *conventional* formal modelling methods which focus on algebraic descriptions of distributed systems. As a consequence, verifying such models in a mathematically rigorous way is often elusive and indeed counter-intuitive when considering established approaches, which generally require sequential process representations or highly abstract theoretical frameworks. The prevalent investigations with this objective in the field of membrane computing are ambivalent and inconclusive in the wider application scope of P systems.

In this thesis we directly address the formal verification of membrane systems by means of model checking. A fundamental distinction between the agnostic perspective on parallelism, advocated by process calculi, and P systems' emblematic *maximally parallel execution strategy* is identified. On this basis, we establish that an intuitional translation to traditional process models is inadequate for the purpose of formal verification, due to a state space growth disparity. The observation is essential for this research project: on one hand it implies the feasibility of model checking P systems, and on the other hand it underlines the suitability of this formal verification technique in the context of membrane computing. Model checking entails an exhaustive state space exploration and does not derive inferences based on the independent instructions comprising a state transition. In this respect, we define a new sequential modelling strategy which is optimal for membrane systems and targets the SPIN formal verification tool.

We introduce elementary P systems, a distributed computational model which subsumes the feature diversity of the membrane computing paradigm and distils its functional vocabulary. A suite of supporting software tools which gravitate around this formalism has also been developed, comprising of 1. the *eps modelling language* for elementary P systems; 2. a parser for the eps specification; 3. a model simulator and 4. a translation tool which targets the Promela specification of the SPIN model checker.

The formal verification approach proposed in this thesis is progressively demonstrated in four heterogeneous case studies, featuring 1. a parallel algorithm applicable to a structured model; 2. a linear time solution to an NP-complete problem; 3. an innovative implementation of the Dining Philosophers scenario (a synchronisation problem) using an elementary P system and 4. a quantitative analysis of a simple random process implemented without the support of a probabilistic

model.

Acknowledgement

I would like to express my gratitude towards my supervisors, Professor Marian Gheorghe and Professor Georg Struth, for their guidance, friendly and valuable advice, unceasing patience and constant support throughout this research project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel computation is a topic of ever increasing significance, powerfully motivated by practical concerns and deeply captivating with its scientific elaborations and philosophical implications. Whilst distributed systems are nowadays ubiquitous, pertaining nearly all technological innovation, research efforts have almost exclusively targeted the *disruptive* effect of concurrency on sequential computation. Formal modelling methods, most notably the family of process calculi, have concentrated on *reconciling* the perceived concurrent, *unregulated* dynamics, with traditional algebraic constructs, used to characterise sequential transitions. These developments were prompted by 1) the necessity for a mathematical framework as basis to formal reasoning in this context, but also 2) to tackle the complexity of sizeable models by systematic construction.

On the opposite front, more audacious and less practically concerned endeavours have advanced different views on parallelism. The quest for effective computability amidst a mutable plurality of entities has sought inspiration and even physical support in natural phenomena and the living world. The abstract models constellated of this insight have originated the field of *natural computing.* Artificial neural networks, DNA computing, cellular automata, computational systems biology and quantum computing are a few of the representative examples in this category. In order to gain a more profound, scientific understanding of parallel dynamics, bio-molecular interactions and other natural phenomena were subject to an unmediated projection into a formal mathematical context. Membrane systems are a typical example of this approach.

A P system (or membrane system) is a parallel computational model inspired by the structure and functioning of the living cell. The calculus performed by P system models is expressed as multiset or string rewriting and communication, and is distributed across a hierarchical structure of compartments. A P system formally captures a very distinctive vision of concurrency in the so called *maximally parallel execution strategy.* This entails that all applicable rewriting and communication rules are executed exhaustively (i.e. for as long as they are applicable) as part of a single, monadic transition.

Formal verification is an aspect generally overlooked when computational models so often described as *unconventional* are considered. Whilst established verification techniques, such as model checking and automated theorem proving, are immediately applicable to distributed systems which map algebraically to process models, it is often cumbersome and indeed counter-intuitive to exercise an existing approach outside its ordinary application scope. The problem is generally reduced to finding an adequate projection to an equivalent sequential model. The difficulty often arises from the incompatibility between the input model required by the verification tool (model checker, theorem prover) and the abstract model subject to examination. The set of primitives offered by a tool's input specification can be misleading; a naive mapping to these may compromise the feasibility of the verification process.

In the context of membrane computing model checking was the preferred formal verification technique, employed in various studies. The outcome of these investigations was mixed and at times, inconclusive. On the one hand, the approach was proved successful for P system models with a single compartment and restricted configurations, and on the other hand, the so called *state explosion problem* was an immediately acknowledged deterrent in circumstances where a better performance was expected. Moreover, the case studies examined are insufficient to infer the feasibility and perhaps non-scalability of model checking P systems, more generally. Whilst there are many clues and very informative results to be drawn from these undertakings, the matter remains irresolute.

In this thesis we directly address the formal verification of membrane systems by means of model checking. The principal objectives and contributions of this research project are to:

1. Identify a key distinction between conventional formal models of distributed systems and membrane computing with its emblematic maximal parallelism;

2. Establish the inadequacy of process models as means of expressing P system transitions for the purpose of formal verification;

3. Introduce elementary P systems, a computational model which a) subsumes the feature diversity exhibited by the membrane computing paradigm and b) distils its functional vocabulary;

4. Define a formal verification approach based on SPIN model checking, optimal for membrane systems (in the context of sequential computation);

5. Design the eps modelling language which allows for an unambiguous and concise representation of an elementary P system;

6. Develop a suite of software tools in support of our formal verification approach, comprising:
   - a parser for the eps specification;
   - a simulator for elementary P systems;
   - an automatic translation tool which takes an elementary P system expressed in the eps notation and targets a sequential Promela model, required by the SPIN model checker.

7. Demonstrate the suitability of our approach on four distinct case studies, featuring:
   - a distributed algorithm for a structured model (counting the child nodes in a DAG);
   - a linear time solution for an NP-complete problem (Subset Sum);
   - an innovative implementation of the Dining Philosophers scenario using elementary P systems;
   - an implementation of a simple random process (a biased coin toss experiment) using a non-probabilistic model.

This thesis is structured as follows:

In chapter 2 we review the key mathematical concepts and data structures pertaining to membrane computing and present P systems extensively, as a distributed and parallel computational model. We next survey the most prominent P system variants with their distinctive features in chapter 3. The fundamental concepts which typify the membrane computing paradigm are identified. The variants which concretise this paradigm are presented with emphasis on expressive power, complexity and application scope, whilst contrasts with the traditional model are also highlighted.

Chapter 4 addresses the notion of formal verification and provides a detailed analysis of the model checking technique. The two supporting formalisms, Büchi automata and temporal logic, are also formally described. The advantages, practicability and inherent limitations of model checking are determined, from the perspective of model checker software tools which automate this technique. SPIN is presented as a leading formal verification tool and in particular, an efficient, open source LTL model checker. Next, a set of prevalent investigations on model checking P systems are scrutinised, highlighting the ambivalence which surrounds this topic. The last section of the chapter reveals an essential incompatibility between the algebraic representation of distributed systems, promoted by process calculi, and the expansive, unitary maximally parallel transition featured by P systems. On this basis, a new sequential modelling approach optimal for membrane systems is introduced. Its principles are outlined in this chapter (4), whilst details of its implementation are discussed in chapter 6.

In chapter 5 we introduce elementary P systems, a computational model rooted in the context of membrane computing which subsumes the diversity and potency of its kindred models. The following chapter describes the formal verification approach by means of SPIN model checking, proposed in this thesis. The set of tools which support this methodology comprises the *eps modelling language* and parser, the *epss* model simulator and the *eps2spin* translation module. These pertain to elementary P system models and are extensively documented in section 6.1. Lastly, chapter 6 exemplifies the approach set forth with the accompanying software tools on a simple P system model which generates numbers from the Fibonacci series.

In chapters 7, 8, 9, 10 we demonstrate the feasibility of our formal verification approach and its suitability over four heterogeneous case studies which target: 1. a structured model; 2. a computationally hard (NP-complete) problem, 3. a synchronisation problem and 4. a non-probabilistic model of a simple random process.

Finally, we present a summary of our findings, concluding remarks and ensuing paths of development in chapter 11.

Chapter 2

P Systems

2.1 Preliminaries

We begin this chapter with a review of key mathematical concepts and data structures pertaining to membrane computing. The notation used throughout this thesis is specified together with any conventions customary to P system models.

A *string* over an alphabet V (a finite and non-empty set of symbols) is a sequence of symbols from V. The empty string is denoted by λ. The length of a string x is denoted by $|x|$ and is equal to the number of symbols it contains. The number of occurrences of symbol $a \in V$ in a string x is expressed as $|x|_a$. Thus, for an alphabet $V = \{a, b, c\}$, a, $ababb$, ccc are strings, with $|ababb| = 5$, $|ababb|_b = 3$ and $|\lambda| = 0$. If an alphabet $V = (a_1, a_2, ..., a_n)$ is ordered and x is a string over V, then $\Psi_V = (|x|_{a_1}, |x|_{a_2}, ..., |x|_{a_n})$ is the *Parikh vector* of x. For example, if $V = (a, b, c)$, then $\Psi_V(ababbc) = (2, 3, 1)$.

A *multiset over* a finite set U of symbols is a mapping $M : U \longrightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. For an element $a \in U$, $M(a)$ represents the *multiplicity* of a in M. The *support* of M is defined as $supp(M) = \{a \in U \mid M(a) \neq 0\}$. If $supp(M) = \emptyset$, then M is the empty multiset, denoted by λ. Typically, finite multisets are expressed as $\{(a_1, M(a_1)), (a_2, M(a_2)), ..., (a_n, M(a_n))\}$, however, in the context of membrane computing, the string notation is preferred and commonly adopted: $w = a_1^{M(a_1)} a_2^{M(a_2)} ... a_n^{M(a_n)}$ and any permutation of w are representations of M ($a_i \in U, 1 \leq i \leq n$). If U is ordered, $U = (a_1, a_2, ..., a_n)$, then the Parikh vector of w is the ordered set of multiplicities $(M(a_1), M(a_2), ..., M(a_3))$.

Given $U = \{a, b, c\}$, then $\{(a, 2), (b, 3), (c, 1)\}$, $\{(b, 2)\}$ and λ are multisets over U. More concisely, the non-empty multisets can be expressed as a^2b^3c (or $c^1a^2b^3$, or b^3ca^2 etc.) and b^2, respectively. Moreover, for an ordered $U = (a, b, c)$, $\Psi_U(a^2b^3c) = (2, 3, 1)$, $\Psi_U(b^2) = (0, 2, 0)$ and $\Psi_U(\lambda) = (0, 0, 0)$.

A graph is an ordered pair (V, E), where V is a finite set of vertices (nodes) and E is a set of unordered pairs from V called edges. The order of a graph (V, E) is given by the cardinality of V, $|V|$, whereas the size is denoted by $|E|$.

2.2 P systems

A P system [77], also known as a membrane system, is a distributed and parallel computational model inspired by the structure and dynamics of the living cell. A P system consists of a hierarchical arrangement of compartments, each having associated a multiset of objects and a finite set of evolution rules.

The key feature of P systems is the so called *membrane structure* which resembles the partitioning of biological entities as generally perceived. The membrane plays the role of a delimiter, and circumscribes a discrete region of computational space. Figure 2.1 illustrates the structure of a P system using a Venn Diagram and a rooted tree, respectively. Membrane 1 is called the skin membrane, it delineates the the outermost compartment and it is identified by the root of the tree.

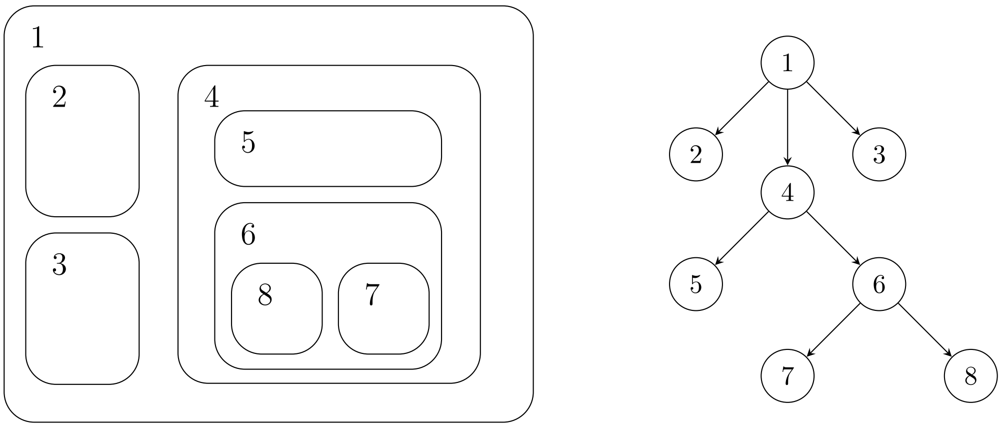

Figure 2.1: A membrane structure example, depicted as a Venn Diagram and a rooted tree. The structure can also be concisely encoded as $[[\,]_2[\,]_3[[\,]_5[[\,]_7[\,]_8]_6]_4]_1$.

Each compartment contains a finite multiset of objects which can be transformed (evolve) into other objects, can pass to neighbouring compartments or can dissolve the membrane they are contained in. These transformations are formally captured by *rewriting*, *communication* and *dissolution* rules, respectively.

A multiset rewriting rule is a transition $u \to v$ where u, v are multisets of objects from the system's alphabet. Such a rule is applicable in a compartment i if and only if there exists a multiset w in i, such that $u \subseteq w$. The execution of this rule results in the subtraction of u from w and the production of v. For instance, $a^3bc^2 \longrightarrow b^2d^4$ takes three a, one b and two c objects and creates two

bs and four ds. The rule is only applicable in compartments containing at least three copies of a, one of b and two of c.

Communication between regions is denoted in rewriting rules by multisets with *target indicators*. These ascribed markers specify which region is to receive the multiset emitted by a compartment. By convention, we refer to a transition of the form $u \rightarrow v_{tar}$, where u, v are multisets of objects from the alphabet O and $tar \in (in, out)$ as a communication rule. If such a rule is executed in a compartment i and $tar = out$, then v is sent to the region outside of i; if $tar = in$, then v is to be passed to one of its direct descendants (inner regions), non-deterministically chosen. A communication rule is applicable in a compartment i if and only if $u \subseteq w$ and, if $tar = in$, then there exists at least one compartment residing in i which can be selected as recipient.

A P system membrane may also be *dissolved*, discharging the multiset content of the compartment it delineates into its parent region. Dissolution is achieved by rules which bear the (reserved) symbol δ, suffixing a (possibly empty) right hand side multiset in a rewriting or communication rule: $a \longrightarrow b\delta$, $a \longrightarrow bc_{out}\delta$, $x \longrightarrow \delta$ are examples of dissolution rules. On this basis, one may consider dissolution as an extension over the basic transition rules having a particular side effect.

One of P systems' most representative features is called *maximum parallelism* and denotes a distinctive rule *execution strategy*. This property entails that *all* transitions which can occur during a computational step, will do so *in parallel*. Whilst compartments demarcate independent computational spaces operating concurrently, P system parallelism persists *inside* membranes, requiring atomic terms, that is, objects or multisets of objects from O, synchronise their transformation. More precisely, all the evolution rules associated with a P system will be executed *simultaneously* and *exhaustively*, in one computational step, across the system's compartments. A more profound treatment of maximal parallelism is presented in Chapter 4.

Formally, a P system of degree $m \geq 1$ is defined as:

$$\Pi = (O, H, \mu, \omega_1, ...\omega_m, R_1, ..., R_m, i_0)$$

where:

1. O is the alphabet of objects;
2. H is the *ordered* alphabet of membrane labels;
3. μ is a membrane structure of degree m;
4. $\omega_1, ..., \omega_m \in O^*$ are multisets of objects associated with the m regions of μ;
5. $R_1, ...R_m$ are finite sets of evolution rules associated with the m regions of μ;

6. $i_0 \in H \cup \{e\}$ specifies the input/output region of Π, where $\{e\} \notin H$ is a reserved symbol which designates the *environment*.

A P system evolution rule can be generically expressed as:

$$v \longrightarrow x y_{in} z_{out} \delta^{?}$$

where v, x, y, z are multisets of objects from O, in, out are target indicators and $\delta \notin O$ is a special marker which can be optionally appended to indicate membrane dissolution (? symbolises zero or one occurrence of δ in the definition above). More specific instances derived from this form resemble the rewriting, communication and dissolution rules described earlier:

- rewriting, when $x \neq \lambda$, $y = \lambda$, $z = \lambda$ and δ is omitted;
- communication, when $y \neq \lambda$ or $z \neq \lambda$ and δ is absent;
- dissolution, when δ is present;

Remark 2.1. Although δ can accompany any rule which is subject to maximally parallel execution, dissolution can only occur once, since this involves the membrane, a singular entity, rather than multisets of objects. Importantly, this does not restrict rewriting and communication rules to a single application, nor does it impose additional constraints on the rule itself.

Since the membrane structure and the multisets associated to each region are the sole volatile elements[1], a P system configuration is generically defined as a tuple which envelops these components:

$$C_{\Pi} = \{\mu, (\omega_1, ..., \omega_m)\}$$

.

A P system transition is defined as the passage between two successive configurations $C \Longrightarrow C'$. A sequence of transitions $C_1 \Longrightarrow C_2 \Longrightarrow ... \Longrightarrow C_h$ is called a *computation* of Π. Naturally, a model which exhibits non-deterministic behaviour will involve multiple such sequences.

A P system computation *halts* when no further rule can be applied in its final configuration C_h. The multiset of objects present in the designated output region i_0 when the system reaches this state, constitutes the result (output) of this computation. If the system's alphabet O is ordered, then the Parikh vector Ψ_{i_0} of the multiset in region i_0 can instead be considered as output.

We next illustrate membrane systems with a simple example (Fig. 2.2).

[1]It is important to highlight that the collection of multisets associated to each compartment of a P system is *not sufficient* to identify a configuration of this system. This is because an empty multiset λ is not equivalent to a dissolved compartment and cannot be used as such.

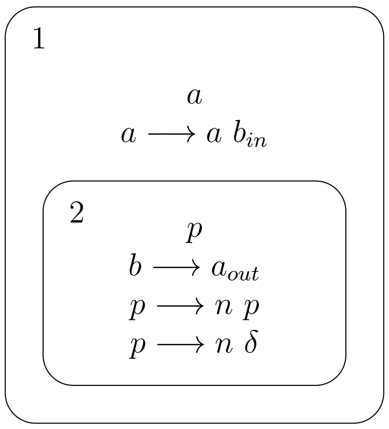
1
a
$a \longrightarrow a\ b_{in}$
2
p
$b \longrightarrow a_{out}$
$p \longrightarrow n\ p$
$p \longrightarrow n\ \delta$

Figure 2.2: Depiction of Π_{Fib} as a Venn Diagram

$$\Pi_{Fib} = (O, H, \mu, \omega_1, \omega_2, R_1, R_2, i_0)$$

where:

$O = \{a, b, p, n\}$,

$H = \{1, 2\}$,

$\mu = [[\ ]_2]_1$,

$\omega_1 = a,\ \omega_2 = p$,

$R_1 = \{a \longrightarrow a\ b_{in}\}$,

$R_2 = \{b \longrightarrow a_{out};\ p \longrightarrow n\ p;\ p \longrightarrow n\ \delta\}$,

$i_0 = 1$.

Π_{Fib} consists of two compartments, labelled 1 and 2, denoting two nested regions. The alphabet O consists of four objects and the initial configuration of the system is $\{\mu, (\omega_1, \omega_2)\}$, where μ is its *initial membrane structure* and ω_1, ω_2 are the initial multisets of regions 1 and 2 respectively. For clarity, we will henceforth refer to the compartments labelled 1 and 2 by σ_1 and σ_2 respectively. The execution of Π_{Fib} proceeds as follows:

Step 1 The presence of one object p enables both rule $p \longrightarrow np$ and $p \longrightarrow n\delta$ in compartment σ_2. This is a standard non-deterministic construct which implicitly prompts for non-deterministic choice. Suppose the former rule is selected and executed, yielding both a p and an n objects. Because p

will not be lost after this step's rule application, the term is conventionally referred to as a catalyst. Since there was only one p, subsequent applications of any of the two rules which require a p are not permissible. The multiset ω_2 does not contain any b objects and consequently rule $b \longrightarrow a_{out}$ is not applicable this step. In compartment labelled 1, rule $a \longrightarrow ab_{in}$ is applied once, assimilating the single a object and producing another a but also a b which is sent to one of its inner compartments. In this particular example, region 2 is the only possible destination for b. Hence, the system's resulting configuration after this first step comprises multisets a and bpn in the two nested compartments σ_1 and σ_2 respectively: $C_1 = \{[[\ ]_2]_1, (a, bpn)\}$;

Step 2 Starting again in compartment σ_2, we assume rule $p \longrightarrow np$ prevails in the non-deterministic selection requiring one p object; p persists across configurations so long as compartment σ_2 remains undissolved and ensures an n is generated each computational step. Since an object b is now available in σ_2, rule $b \longrightarrow a_{out}$ can and *must* be applied, emitting a single a to its parent region 1. Rule $a \longrightarrow ab_{in}$ is executed once in compartment 1, replicating a and inserting a b into its only child compartment - a transition identical to the one in the preceding step. The resulting configuration is $C_2 = \{[[\ ]_2]_1, (a^2, bpn^2)\}$;

Step 3 We apply the same rules selected in step 2 for compartment σ_2 in exactly the same way, since the multiplicities of b and p are unchanged - implicitly, we have also maintained the non-deterministic choice resolution, that is, in favour of the non-destructive rule. In compartment σ_1 on the other hand, the communication rule $a \longrightarrow ab_{in}$ will be applied two times, yielding two as in the respective region and two bs sent to σ_2. $C_3 = \{[[\ ]_2]_1, (a^3, b^2pn^3)\}$;

Step 4 We consider another step during which rules are executed analogously to *Step 3*; $a \longrightarrow b_{in}$ is applied three times whereas $b \longrightarrow a_{out}$ expends the two bs and sends two as to region 1. $C_4 = \{[[\ ]_2]_1, (a^5, b^3pn^4)\}$;

Step 5 At this point, we suppose $p \longrightarrow n\delta$ is selected in region 2, the effect of which is to dissolve the compartment and release its content to the parent region 1. This is accomplished *after* all rewriting and communication rules have been applied across the two compartments. More precisely, the five bs emitted to σ_2 as a result of rule $a \longrightarrow ab_{in}$ are finally incorporated into σ_1, despite the fact they were produced during the same step. Naturally, as soon as a compartment is dissolved, its associated rules become irrelevant. Since the single rule in compartment σ_1, $a \longrightarrow ab_{in}$, requires a recipient to send a b object, it too becomes inapplicable when region 2 is invalidated. Consequently, the computation halts at this step with the resulting final configuration $C_5 = \{[\ ]_1, (a^8b^5n^5)\}$. Compartment σ_1 is also the designated output region of Π_{Fib}, as indicated by i_0 which entails that multiset $a^8b^5n^5$ is the output of computation $C_1 \Longrightarrow C_2 \Longrightarrow C_3 \Longrightarrow C_4 \Longrightarrow C_5$.

Table 2.1 illustrates the outcome of each computational step described above, for convenient scrutiny.

Step/Compartment	σ_1	σ_2
0	a	p
1	a	bpn
2	a^2	bpn^2
3	a^3	b^2pn^3
4	a^5	b^3pn^4
5	$a^8b^5n^5$	#

Table 2.1: Listing of a five step computation of Π_{Fib}

Observation 1. Each computational step Π_{Fib} generates the *next* number in the Fibonacci series, encoded as the multiplicity of object a in compartment σ_1.

Observation 2. The system Π_{Fib} may never halt. The only possible transition to a final configuration, determined by rule $a \longrightarrow n\delta$, is subject to non-deterministic choice.

Observation 3. Every Π_{Fib} halting computation outputs a multiset m, such that the following relation *always* holds: $Fib(|m|_n) = |m|_a$, where Fib is a mapping between a natural number k and the kth element in the Fibonacci series, and $|x|_y$ denotes the multiplicity of symbol y in multiset x.

P systems are computationally universal models. This has been initially demonstrated in [77] by simulation of *matrix grammars with appearance checking* [34]. It was shown that P systems with two membranes containing rules with priorities and one catalyst are sufficiently expressive to generate the recursively enumerable sets of natural numbers. Subsequent attempts [69, 84] proved, by reduction to *register machines*, that priorities for P system evolution rules are not required to achieve universality.

P systems, with their suite of primitives but also the transitions employed, are evocative of other discrete, distributed models. Petri Net [71] transitions are also described in relation to a structured computational space, which is that of a bipartite graph, and operate concurrently on multiplicities of atomic *tokens.* Cellular automata [85] feature a grid structure of cells which execute a fixed set of instructions simultaneously. This resembles the the finite set of evolution rules ascribed to P system compartments which are also applied in parallel.

Another parallel rewriting model which appears kindred with membrane systems is L-systems [62]. Inspired by the growth patterns of various types of algae, L-system grammars were introduced to formally describe multicellular developments. A most noteworthy trait of L-systems is the *exhaustive* execution of production rules. L-system rewriting rules operate on and generate strings; these differ from P system evolution rules whose subject is the multiset, but may also include the compartment the rules are applied in (dissolution rules).

Whilst a deeper analysis of the similarities and contrasts between related models may be an interesting undertaking, within the scope of this thesis it is sufficient to acknowledge the contexts membrane systems are situated in, namely:

1. nature inspired formal models and
2. distributed and parallel computational models.

We anticipate, the most pertinent distinction is to be identified between P systems and process algebra such as CSP and π-calculus whose reconciliatory treatment of parallelism exerted a resolute, almost exclusive influence on the formal verification of distributed systems.

Ever so frequently said to be inspired by the living cell, P systems are not simply an indeterminate, linear abstraction of biological entities and their inherent transformations, but rather a powerful synthesis of concepts and ideas with profound influences. Without trespassing into a philosophical domain, we can infer that membrane systems are the scientific expression of an organic vision of dynamics. All transient entities envisaged are consubstantial *in their transformation.* A transition is not an individualized and isolated event, proper to a singular process, but rather a monadic transmutation to which entities, in their plurality, collectively *participate.*

It is this rather stylistic observation which is not only relevant but indeed consequential in our interpretation of parallelism and the sequential modelling approach set forth in this thesis (Chapter 4).

Chapter 3

The membrane computing paradigm

Since its emergence in 1998 [77], membrane computing has received remarkable and sustained attention, resulting in a proliferation of computational models which gravitate around a central concept. These developments do not constitute an evolutionary stage (in the strict sense) in this branch of natural computing, the traditional P system model has not been superseded nor does it stand as a prototype. In an effort to characterise this flux of realisations in its distinctive occurrence, perhaps it is not inadequate to speak of *model transfiguration* in an abstract formal context. Each P system variant is defined as a unique arrangement of primitives and novel semantics, widening the application scope or aggregating functional elements more concisely, but never dissociates itself categorically from its stem. Rather, a model can be viewed as an individualised embodiment of a computational paradigm. Collectively, P system models 1) *concretise* the *membrane computing paradigm* and, in their diversity, 2) *substantiate* it at the same time.

We proceed with a survey of the most prominent ideas and developments in the field on membrane computing. The principal objective of this examination is to identify the models' distinctive functional components, their expressive power and complexity on the one hand, but also the presence and permanence of key concepts which typify the computational paradigm instantiated. In this respect, more detailed descriptions and remarks on the novelties, advantages and potential incongruities are preferred over complete reproductions of mathematical definitions (references to the relevant works are provided).

Proofs of universality for models, their application scope and efficiency in tackling specific problems are also highlighted. Several examples have been selected to illustrate the model semantics more clearly, but also to demonstrate the noteworthy algorithms which underpin a P system variant, some of which are revisited and re-imagined in our case studies. A discussion on how this cluster of computational models can be interpreted will conclude this chapter.

In terms of notation, we use lower-case letters to denote objects (elements) of a set or multiset and upper-case for set identifiers. A P system's alphabet of objects is symbolised by O, whereas compartments are referred to by the Greek letter σ and a subscript $i \in \mathbb{N}$. A set of transition rules is generally denoted by R, or R_i if the rules are bound to a compartment σ_i. The initial multiset of objects present in a compartment σ_i is marked as ω_i.

3.1 P systems with permitting and forbidding contexts

All traditional evolution rules (multiset rewriting, communication and dissolution) featured by transition P systems are inherently conditional instructions since they depend on the presence of the left hand side multiset in the compartment they are executed. Communication rules also require an eligible recipient region for each multiset with a target indicator. Whilst these innate constraints are sufficiently powerful to express more complex conditions over a number of auxiliary steps, a simpler alternative was propounded whereby designated multisets can be appointed to regulate the execution of rules. These multisets are generally referred to as promoters (or activators) and inhibitors. A rule of the form $a \longrightarrow b|_{z,\neg t}$ residing in a compartment i with contents w_i (a, b, z, t, w_i are multisets over the alphabet objects O), is applicable if and only if 1. $z \subseteq w_i$ and 2. $t \not\subseteq w_i$ and 3. $a \subseteq w_i$. The third of the predicates is generic to rewriting rules and is always re-evaluated as the multiset w_i depletes, allowing for an exhaustive execution. The first two conditions which relate to z and t respectively are assessed prior to any other requisite and their validity is sustained for the entire computational step. Hence, a rule $a \longrightarrow b|_{a^3}$ will execute at least three times, provided the number of as in the compartment is *initially* greater or equal to three and no other rule expends a objects. Conversely, if $|w_i|_a < 3$ at the beginning of the step, the rule is inapplicable. As underlined in this example, the multiset a^3 is interpreted as a shallow guard over w_i, the multiset content of compartment i. The mechanism is shallow or static because its evaluation does not ripple any changes in w_i (which is the subject operand in the comparison), it is not considered for repetitive executions of the rule and cannot be modified in response to computational developments during subsequent steps.

Rules with promoters and inhibitors were extensively used in P system based static sorting algorithms [13, 16, 26], in conjunction with symport/antiport communication which is described in the following section.

3.2 P systems with symport/antiport communication

Whilst the traditional rewriting and communication rules confer ample computational power on P systems to achieve universality, an interesting question was posited in an attempt to identify a more restrictive set of primitives for this purpose: are basic communication rules alone sufficiently expressive transitions to simulate Turing machines? This perhaps foreseeable inquiry, given the distributed nature of P system models, was decisively addressed in a study, revealingly titled *The Power of Communication: P Systems with Symport/Antiport* [75]. Symport and antiport rules mathematically abstract the transfer of chemicals through cellular membranes:

- When two chemicals pass together into or out of a region, the process is referred to as *symport* and can be formally expressed as (ab, in) and (ab, out) respectively, where a and b are objects from a P system alphabet O;
- Chemicals which pass in opposite directions symbiotically are said to operate in *antiport*, defined as $(a, in; b, out)$, where $a, b \in O$;

The single object transfer (a, in) or (a, out) is also considered in this context as a *uniport* rule.

Symport (and implicitly uniport) rules are applicable in a P system compartment if the objects required to complete the transfer are available, that is, included in the compartment's inner multiset. Antiport rules elevate P system communication to *synchronised bidirectional transfer* and consequently require the respective objects be present inside and outside the compartment the rule is evaluated in. Specifically, if $(a, in; b, out)$ is a rule of compartment σ_i, then at least an object b must be present in σ_i and at least an a must exist in the region containing σ_i in order to apply this rule.

It was demonstrated in [75] that P systems with symport/antiport rules and minimum five membranes are capable of simulating *matrix grammars with appearance checking* [35] and hence computationally universal. Furthermore, it was shown that the number of membranes can be reduced to two if complex antiport rules, such as $(ab, in; cd, out)$, are utilised.

We next illustrate symport/antiport rules together with promoter objects in a static sorting algorithm presented in [26]. The P system employed comprises n nested membranes $[...[[]_1]_2...]_n$, where n is the number of positive integers to be sorted. Each integer is encoded as the multiplicity of an object a present in a compartment $\sigma_i, 1 \leq i \leq n$. During a computational step, a pair of rules referred to as a *comparator* is applied between neighbouring compartments alternately, until the system reaches a *stable configuration over a*, that is, the distribution of a objects accross the system remains unaltered by any subsequent rule application. At this point, the n numbers are ordered with the lowest integer residing in the outermost region and implicitly, the highest integer pushed to the innermost compartment.

The two rules which effectively perform a parallel swap operation between unsorted integers in adjacent compartments can be formally expressed as:

$$R_1 = \{(a, in; a, out)|_p > (a, in)|_p\}$$

where $a \in O$, p is promoter and the $>$ symbol indicates a priority relation. Specifically, the antiport rule, if applicable, must always precede the symport rule. Both rules are guarded by an activator p whose absence prohibits their execution. By allowing promoters inside either odd or even compartments at one time, the comparator rules are coordinated across adjoining compartments and achieve the goal of transferring the higher multiplicity to the inner region. The alternating application of these rules characterises the P system algorithm, which is often referred to as *odd-even sorting*.

To enable the promoter p to switch between odd and even compartments, the uniport rules (p, in) and (p, out) are essential. These supplement the aforementioned comparator rules for odd compartments, establishing the following sequence:

$$R_2 = \{(p, in) > (a, in; a, out)|_p > (a, in)|_p > (p, out)\}$$

The priority relation denoted by $>$ does not impede rules of a lower rank (in such a sequence) from executing if a high priority rule is inapplicable - this is generally regarded as the *weak interpretation* of fixed priorities in the context of membrane computing. If there is no p to be sent to a compartment σ_i via rule (p, in), then the antiport rule $(a, in; a, out)|_p$ can still be executed in σ_i, provided the rule conditions are satisfied (there is at least one a inside the compartment and an activator p is present)[1].

Figure 3.1 depicts the initial configuration of a static sorting P system example with an array of $n = 6$ natural numbers, $[5, 3, 8, 4, 1, 7]$. These are distributed as multiplicities of object a across six nested compartments as follows: $\omega_1 = a^7, p$, $\omega_2 = a^1$, $\omega_3 = a^4, p$, $\omega_4 = a^8$, $\omega_5 = a^3, p$, $\omega_6 = a^5$. We underline that ω_1 represents the multiset of the innermost compartment, σ_1; in our schematic depiction, each

[1]In this scenario, it is not strictly required to impose a priority between the rules affecting promoter p, since there is only one instance of this object shared between two adjacent compartments. More precisely, a single rule of this type can be executed per computational step. We have opted, however, for a faithful reproduction of the odd-even sorting algorithm introduced in [26].

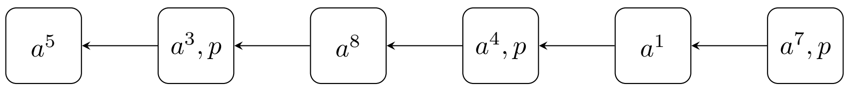

Figure 3.1: A set of six nested compartments depicting the membrane structure of our odd-even sorting example. The arrows denote a child-parent relationship between compartments.

Step/Compartment	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	a^7, p	a	a^4, p	a^8	a^3, p	a^5
1	a^7	a, p	a^8	a^4, p	a^5	a^3, p
2	a^7, p	a^8	a, p	a^5	a^4, p	a^3
3	a^8	a^7, p	a^5	a, p	a^4	a^3, p
4	a^8, p	a^7	a^5, p	a^4	a, p	a^3
5	a^8	a^7, p	a^5	a^4, p	a^3	a, p
6	a^8, p	a^7	a^5, p	a^4	a^3, p	a

Table 3.1: The execution trace of an odd-even sorting algorithm based on P system with symport and antiport rules.

rectangle denotes a compartment with the specified multiset content, whereas the arrows suggest containment and point towards the parent compartment.

Each odd membrane is associated rule set R_2, whereas R_1 is sufficient for the even compartments. The outermost compartment (σ_n) does not require any rules, regardless of n being odd or even[1].

The odd compartments will commence execution since they contain one p which activates the comparator rules. The uniport rule (p, in) is skipped since there are no external ps to transfer inside the membrane. The antiport rule exchanges k copies of a between compartment pairs (σ_i, σ_{i+1}), $1 \leq i \leq n-1$, where $k = min(|a|_{\sigma_i}, |a|_{\sigma_{i+1}})$, that is, the minimum between the two multiplicities. The remaining a objects in σ_{i+1} (if any), will be absorbed by σ_i, using the uniport (a, in). Finally, p is sent *out* to the parent compartment which is to perform the same exchange with its own parent (left aligned compartment in figure 3.1) during the next computational step.

Table 3.1 lists the execution trace (i.e. the system's configurations) for the six steps required to sort the encoded integers. In this example, it can be observed

[1]The outermost compartment *must* never include a promoter p in its initial state if $n = 2k + 1$.

that after five steps the P system reaches a stable configuration on a, which is the stipulated halting condition. Whilst we can generally affirm that a maximum of n steps are necessary to sort n natural numbers using an odd-even sorting P system (indeed this is the concluding remark in [26]), the computational time T required can be expressed more accurately in terms of the offset of an integer relative to its index in the sorted sequence. More precisely, the system requires a number of steps that is equal to the maximum offset found in the array. If a number x is situated at index i in the initial sequence (configuration), and its location in the sorted array is at index j, then its offset is $\Delta_x = abs(i - j)$, where abs stands for the absolute value of a number. On this basis, T equates to $max(\Delta_{x_k}), 1 \leq k \leq n$ computational steps.

Symport/antiport communication has been a pivotal development, deeply influential and persistently investigated in the context of membrane computing. Whilst the computational power (Turing completeness) of P systems with symport/antiport rules was independently demonstrated in various configurations [42, 43, 46], the rules have also been incorporated into other variants (as we will see next) and studied in conjunction with alternative execution strategies, notably minimal parallelism [29, 45].

3.3 Tissue and Population P systems

P system transitions are enacted in a structured computational space. The original model, inspired by the eukaryotic cell, envisaged a hierarchical arrangement of chambers which identify separate regions in this space. Whilst a *tree-like* structure of compartments is a perfectly suitable environment for maximally parallel transitions and concurrent computation in general, it appeared restrictive and inflexible when P systems were considered as a modelling framework in other scientific contexts. Indeed, not all entities (whether organic or inorganic, macro or micro) are organised in a precisely determined structure and it is often the case that *structural change* is intrinsic to the abstract model they are subject to. Moreover, the ability to devise a concise and intuitive mapping is a decisive consideration[1] which effectively requires a one to one correlation between P system compartments and physical entities - the individuals of a colony for instance.

Tissue P systems [63] address these concerns by extending the tree-like membrane structure to that of a graph. Whilst this may be perceived as a generalisation of the structure of compartments (i.e. a tree can be thought of as a specific

[1]Since P systems are computationally universal, they can model any deterministic dynamic which can be computerised. This, however, may require elaborate encodings over multiple objects and across several compartments depending on the complexity of the subject being modelled. The necessity to introduce intricacy in a model is (almost) never desired.

directed acyclic graph where each node, excepting the root, has a single parent), it is important to underline that tissue P systems do not represent a general form of the traditional model. Since the target indicators (*in*, *out*) are no longer applicable, communication rules in tissue P systems have been associated with *communication channels*, symbolised by the *edges* between nodes, as opposed to compartments. This approach, a hallmark of tissue P systems, allows for a concise design of object exchange between connected compartments, requiring no more than the set of channels which constitutes the membrane structure.

Initially, tissue P systems were investigated as purely communicative models [76], however a series of extensions followed. We examine one of the original (and prevalent) definitions, that is of tissue P systems with channel-states, and iterate over the related developments that have branched off this model.

A tissue P system with channel states and symport/antiport rules [44] is a construct:

$$\Pi = (O, K, \omega_1, ..., \omega_m, E, ch, (s_{i,j})_{(i,j)\in ch}, (R_{(i,j)})_{(i,j)\in ch}, i_0)$$

where O is the finite alphabet of objects, K the set of channel states, $w_1, ..., w_m$ are finite multisets of objects from O initially present in the respective compartments $1, ..., m$; E denotes the finite set of objects arbitrarily available in the *environment*, which is usually labelled 0; $ch \subseteq \{(i,j)|i,j \in \{0,1,...,m\}, i \neq j\}$ is the set of channels between compartments, or between a compartment and the environment. Each channel symbolised by the ordered pair (i,j) is associated a state $s_{(i,j)} \in K$ and a finite set of symport/antiport rules $R_{(i,j)}$. A rule from $R_{(i,j)}$ is of the form $(s, x/y, s')$, with $s, s' \in K$, and $x, y, \in O^*$. Accordingly, such a communication rule has the capacity to change the state of the channel in addition to exchanging objects between compartments (or with the environment). i_0 designates the output compartment of Π.

The usage of states on communication channels confers versatile conditioning for symport/antiport rules and generally allows for a more controlled execution. There is, however, a pronounced incompatibility which prompts for a restriction (or other reconciliatory measures) when considering the maximally parallel execution strategy. Since maximal parallelism requires all applicable rules be executed exhaustively, there may be rules on a channel $R_{(i,j)}$ contending to change the state $s_{(i,j)}$ of that channel to different values during the same step. This was resolved in various instances by applying only rules which solicit a transition to the same state (non-deterministically chosen). However, a more conventional alternative was also considered involving an adjustment on the execution strategy: *maximal parallelism with sequential behaviour on channels* [44] requires that at most one rule per channel per step is applied. A more stringent provision still is the *sequential mode* [41], which imposes that a single channel and a single rule are non-deterministically selected during each computational step. This was investigated in antithesis with the *asynchronous* [41] strategy according to which an arbitrary number of channels and rules (for the selected channels) can be active at each step. The referenced studies are instructive of how various execution strategies pertain to tissue P systems and reinforce the modelling potential of this variant. Complementary, the model's computational power and complexity have also been extensively investigated [9–12]. Specifically, it was demonstrated that tissue P systems with simple antiport rules or symport rules of size two in two compartments are sufficient to achieve universality.

The execution strategy was not the sole element to induce variation in tissue P systems. Substantial research also focused on developments which include rewriting rules alongside symport/antiport communication. The models defined in this respect are generally referred to as *Evolution-Communication* (or EC) P systems and constitute a class of their own (we investigate one prominent variant in more detail, in section 3.5). Additionally, tissue P systems with string objects and rewriting rules have also been examined [66], motivated by the modelling requirement of more complex macromolecules, such as DNA and RNA, which are usually depicted as strings.

Population P systems [19] are a noteworthy extension of tissue P systems and its principal objective was to allow structural mutation. Whilst the edges of a graph appeared as an intuitive abstraction for communication channels between more complex biological entities (but not exclusive), these were not sufficiently expressive to succinctly model the behavioural dynamics of societies or metapopulations. Population P systems introduced *bond making* rules to allow the creation of new channels during a computational step. A bond making rule is denoted as $(i, x; y, j)$ and its application results in a new undirected communication channel between two compartments i and j, when i contains at least one object x and $j \neq i$ includes a y.

Population P systems employ a maximally parallel execution strategy for rewriting and communication rules. During each computational step, however, all connections (edges in the graph structure) are removed and re-instated as stipulated by the applicable bond making rules. Hence, the membrane structure is transformed in parallel with any other executable instruction and, as a mutable collection of elements, it is included in a system configuration C_i for a step i. Since there are no link destruction rules, no conflicting scenarios are identified - the repeated execution of a bond making rule is equivalent to a single execution and hence, inconsequential.

Interestingly, one may consider tissue P systems with channel states capable of emulating a dynamic structure by alternation of two states, one corresponding to an *active* channel and its complement, the *inactive* state. In this respect, communication rules can be prohibited execution if the channel is in an inactive state. Bond making rules, however, are applied independently of any other rule and are only conditioned by the presence of multiset in the two designated compartments (i, j in the above expression). Similarly to promoter objects, the multisets required are not consumed when the rule is applied.

A noteworthy case study based on population P systems (with slight variations) investigates a model of quorum sensing behaviour in a Vibrio fischeri bacterium [20]. Additionally, we mention the modelling principles presented in [21] for describing metapopulations using P systems as formal framework.

3.4 Spiking neural P systems

Spiking Neural P system (SN P systems), introduced in [54], directly abstract from the nervous cells (neurons) and their interactions, particularly the communication by means of electrical impulses of identical shapes, called *spikes*. Whilst they preserve the same internal structure as their kindred Tissue P system variant, the graph which represents the set of synapses between neurons is directed and generally referred to as *synapse graph*. Another notable particularity is the singleton alphabet $O = \{a\}$, where a can be associated a multiplicity and likewise borrows the biological term *spike*. There are two types of rules which can be applied by a neuron σ_i: (a) *firing rules* of the form $E/a^c \longrightarrow a; d$ and (b) *forgetting rules* expressed as $a^s \longrightarrow \lambda; d$. A firing rule can be guarded by a regular expression E over a and as such, may only execute if $a^c \in L(E)$ (the regular language defined by E).

One element which further underpins the model's faithfulness to neural physiology is the constant d which accounts for the so called *refractory period* of neurons, during which the cell cannot emit or receive spikes and is thus considered closed or inactive. d can be attached to both rules and indicates the number of computational steps the cell becomes closed for, after the respective rule was applied.

A firing rule, when executed inside a neuron σ_i, removes c spikes from σ_i and emits *one* spike to *all* forward connected neurons σ_j, $(i, j) \in syn$ (the set of synapses). A firing rule propagates electrical spikes to neighbouring compartments independent of the number spikes required on the left hand side; it is the number of recipients which determines the output of the rule. Notably, Spiking Neural P systems is the first model to feature such rules, where communication is envisaged as a radial dispersal of discrete entities, as opposed to correlated inter-compartmental rewriting exercised by cell-like and tissue P systems; the concept will complement traditional multiset rewriting in a more complex development, described in the next section.

A forgetting rule on the other hand can only execute in σ_i if σ_i contains precisely s spikes and no firing rules are applicable. It operates by removing s spikes from σ_i.

SN P systems were persistently investigated as language recognisers and generators [27, 28] but also as efficient computational devices for solving intractable (NP-complete) decision problems, such as Subset Sum [60, 61], SAT and 3-SAT [57].

3.5 Neural and Hyperdag P systems

Neural P systems (nP systems) [69] constitute the first P system class to syncretise a semantically eclectic mix of elements advanced by other variants. The model is noteworthy for its orchestration of multiple execution strategies and augmented communication scope. Specifically, nP system compartments are assigned a *process mode* $\alpha \in \{min, par, max\}$ and a *transfer mode* $\beta \in \{one, repl, spread\}$. The *min* and *max* labels correspond to *minimal* and *maximal* parallelism respectively, strategies described in the preceding chapter. The *par* directive allows *arbitrary* executions of an *arbitrary* selection of applicable rules in a compartment. Importantly, both the process and transfer modes are fixed instructions and cannot be changed (for nP systems) as the system transitions to new states.

The $\beta = one$ mode accounts for traditional communication rules which emit a multiset m of objects from the alphabet O to *one* recipient only. Since Neural P system compartments are linked by directed arcs (the membrane structure resembles a directed graph), a target compartment is non-deterministically selected when more than one prospective recipient are available. By contrast, the *repl* tag requires the emitted multiset m be *replicated* to all *forward* connected compartments. Finally, *spread* induces non-deterministic division of m into *sub-multisets* which are dispersed to arbitrary neighbours (outward linked compartments).

In addition to individual execution strategies and transfer modes, nP system compartments are also attributed a set of states Q such that at each computational step, only rules associated with the state $s \in Q$ the compartment is in are applicable. More formally, a rule can be expressed as $sx \longrightarrow s'x'y$, where $s, s' \in Q$, $x, x' \in O^*$ and $y \in O$ is a multiset sent to neighbouring compartments according to the transfer mode β of the compartment the rule executes in. Notably, a rule, if applied can also change the state of its compartment concomitantly, however, this transition ($s... \longrightarrow s'...$) is always singular and not regulated by the process mode α.

A single system compartment can be designated as the *output cell*, which can solely send objects *outside* the computational context (generally referred to as the *environment*). The system halts when no further transitions are possible (i.e. a deadlock is reached) and the result of the computation is the Parikh vector of the multiset accumulated in the system's environment.

Neural P systems, whilst not a prominent formalism with immediate applications, have had a catalytic influence in the emergence of other elaborate variants which aim to integrate heterogeneous primitives, in particular Kernel P systems [48]. Moreover, the model provides the foundational basis for Hyperdag P systems (hP systems) [80, 82], an extension primarily investigated for its modelling potential and expressive power in the context of computer networking. By constraining the model's membrane structure to that of a directed acyclic graph (DAG), communication between compartments can be channelled effectively, without resorting to contrived implementations. A series of network discovery algorithms devised with hP system primitives have demonstrated [81] efficacy and intuitive design. The principal addition which accompanies the DAG structure is the directional label affixed to multisets in communication rules. Since DAG nodes (vertices) can be ordered alongside an axis, three distinct relations can be derived from this arrangement: *parent-child*, *child-parent*, *sibling-sibling*. These relations refine Neural P system communication rules, allowing a more focused replication and transfer of objects from a compartment. Formally, an hP system rule can be expressed as: $sx \longrightarrow s'x'u_{\uparrow}v_{\downarrow}w_{\leftrightarrow}y_{go}z_{out}$ where:

- s, s' are states from the set of states Q;
- x, x' are multisets over the alphabet O, essentially the constituents of a rewrite operation;
- $u_{\uparrow}$ denotes a multiset destined to parent compartments;
- $v_{\downarrow}$ is a multiset which is to be sent to child compartments;
- $w_{\leftrightarrow}$ is a multiset which targets sibling compartments;
- y_{go} is a multiset which could be received by all connected compartments;
- z_{out} denotes a multiset to be expelled into the system's *environment*;

Each multiset emitted to neighbouring compartments, as indicated by its associated tag, will respect the transfer mode β which now applies on the restricted set of compartments. Specifically, if $\beta = repl$, then a multiset $u_{\uparrow}$ will be replicated to all *parent* nodes, $v_{\downarrow}$ would be replicated to all *child* nodes and analogously for $w_{\leftrightarrow}$, y_{go}. Conversely, if $\beta = one$, then only one of the respective nodes (parent, child, sibling) will be non-deterministically chosen as the recipient.

The so called *process mode* is also retained and can be associated with states from Q in addition to compartments. More precisely, a process mode need not be fixed from the start (the initial configuration) and operates dynamically in hP

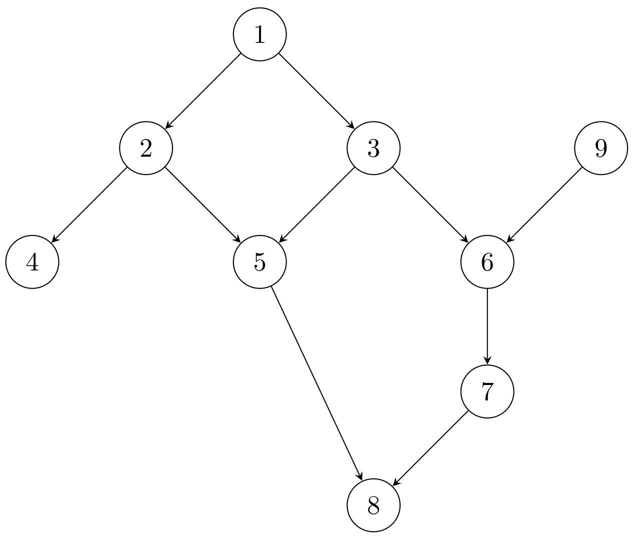

Figure 3.2: A directed acyclic graph example. The *child count via broadcasting* hP system algorithm is demonstrated on this configuration.

systems where state transitions may switch between the three execution strategies in β.

We further illustrate Hyperdag P systems with an example (adapted from [81]) which will be revisited and elaborated into a dedicated case study.

We consider an hP system with alphabet $O = \{a\}$, compartments $\sigma_1 ... \sigma_9$ arranged in a DAG structure, depicted in figure 3.2, where a numbered vertex i indicates compartment σ_i and each arrow represents a directional link between nodes. Each compartment is associated the same set of rules:

1. $s_0 a \longrightarrow s_1 p_{\downarrow}$, with $\alpha = min$, $\beta = repl$;
2. $s_0 p \longrightarrow s_1 p_{\downarrow} c_{\uparrow}$, with $\alpha = min$, $\beta = repl$;
3. $s_1 p \longrightarrow s_1$, with $\alpha = max$.

All nine compartments start in state s_0 with root nodes (i.e. nodes without a parent) σ_1, σ_9 including a non-empty multiset a. Each compartment awaits an object p which prompts an acknowledgement c to all its parent nodes. The same rule further propagates p to child compartments and performs the transition to state s_1 at which point any remaining p objects will be *destroyed* (using rule 3) since they are no longer relevant. We note the transfer mode $\beta = repl$ for the first two rules which indicates that objects are always broadcast to all connected nodes in the respective direction. Additionally, $\alpha = min$ prevents multiple executions of the second rule in case two or more p objects are received simultaneously - in

Step/Compartment	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8	σ_9
0	s_0a	s_0	s_0	s_0	s_0	s_0	s_0	s_0	s_0a
1	s_1	s_0p	s_0p	s_0	s_0	s_0p	s_0	s_0	s_1
2	s_1cc	s_1	s_1c	s_0p	s_0pp	s_1p	s_0p	s_0	s_1c
3	s_1cc	s_1cc	s_1cc	s_1	s_1	s_1c	s_1	s_0p	s_1c
4	s_1cc	s_1cc	s_1cc	s_1	s_1c	s_1c	s_1c	s_1	s_1c

Table 3.2: The execution trace of the *child count via broadcasting* algorithm applied on the DAG depicted in figure 3.2

our example, this occurs at step 2 in compartment σ_5. Conversely, the maximal parallelism assigned to rule 3 disposes of all superfluous ps in one step.

The algorithm completes when all compartments have reached state s_1 and the remaining p objects have been cleared. This is achieved after a maximum $h + 1$ steps, where h is the height of the graph. In its final configuration, each node (compartment) will contain a number of cs that is equal to the number of its child nodes.

We conclude our treatment of Hyperdag P systems with a remark evinced by the presented example: hP system communication rules do not require an existing recipient to execute, the left hand side multiset condition common to all P system variants, together with the state guard are the only requirements to be satisfied. Rule 2 ($s_0p \longrightarrow s_1p_{\downarrow}c_{\uparrow}$) executes in compartment σ_8 despite the fact there are no child nodes to send p to.

3.6 P systems with active membranes

P systems with active membranes [78] extend the maximally parallel dynamics of traditional models, onto the membrane structure itself. Membrane creation and division naturally complement the dissolution rules transition P systems are endowed with, and provide means of expanding the computational space to accommodate new multiset *instances* over the same alphabet. Formally, division rules are expressed as $[a]_h^{e_1} \longrightarrow [b]_h^{e_2}[c]_h^{e_3}$, where a, b, c are multisets over a system alphabet V, h is a membrane label and $e_1, e_2, e_3 \in \{+, -, 0\}$ represent *electrical charges* bound to membrane h. Applying such a rule transforms multiset a and its parent compartment on the left hand side into two new compartments with the same label h containing multisets b and c respectively and possibly different polarization (e_2 and e_3). Furthermore, the existing contents of the compartment initiating the division is replicated to the nascent regions signified by the right hand side of the rule.

P systems with active membranes employ a conventional tree like structure of compartments, however, the concept of electrical charges or polarization was also introduced (as specified in the above definition) in order to distinguish between three states a membrane can situate in. Communication and division rules have the capacity to change the polarization of a membrane and hence determine subsequent execution. A rule which stipulates an electrical charge e may only be applied in a compartment with label h if h bears the charge e: $[a]_1^+ \longrightarrow []_1^- a$ is a communication rule which sends one a outside compartment 1 and changes the polarization $+$ to $-$ concomitantly; the rule can only be applied when membrane 1 has the positive charge $+$.

The single most important quality of this P system variant is its ability to *generate an exponential computational space in linear time.* This is achieved by executing the emblematic membrane division rules concurrently: every step, a compartment may be divided[1] and its contents replicated and/or mutated in one atomic procedure. This approach, frequently referred to as "trading space for time", was employed extensively and almost exclusively for solving computationally hard problems which *currently*[2] require exponential resources (in space, time or both). Linear time solutions have been devised for all well known NP-complete decision problems, demonstrating the efficacy of maximal parallelism in conjunction with membrane creation: SAT, 3SAT [47, 67, 78], the Knapsack problem [58, 68], 2-Partition [50], 3-COL [37, 49], Subset Sum [36, 39, 70].

We illustrate the exponential growth of compartments by membrane division in figure 3.3, which depicts the first 4 computational steps of a Kernel P system solution to the Subset Sum problem we presented in [39]. The outlined instance of the problem considers a set $A = \{3, 8, 25, 23, 5, 14, 30\}$ of positive integers with the stipulated sum $S = 55$. Each level in the hierarchy represents a configuration of the system at a given step. We also note the dashed arrows symbolize *division* in this diagram and not the *parent-child* relationship between compartments in a tree-like membrane structure. Each number is represented by the multiplicity of the object x, whilst the presence of a indicates the compartment is active and no answer has been found on the respective path. The approach is described in detail and formally documented in [39].

Notably, division rules for their powerful semantics also disturb the regularity of the maximally parallel execution strategy, at least in its pure interpretation. Division rules operate on compartments, a consideration which entails the fol-

[1]Division rules generally transform one compartment into two disjoint regions, however extensions to this model have investigated the possibility and potential of an arbitrary expansion of the computational space per step.

[2]At the time of writing, the $P = NP$ conjecture is unresolved.

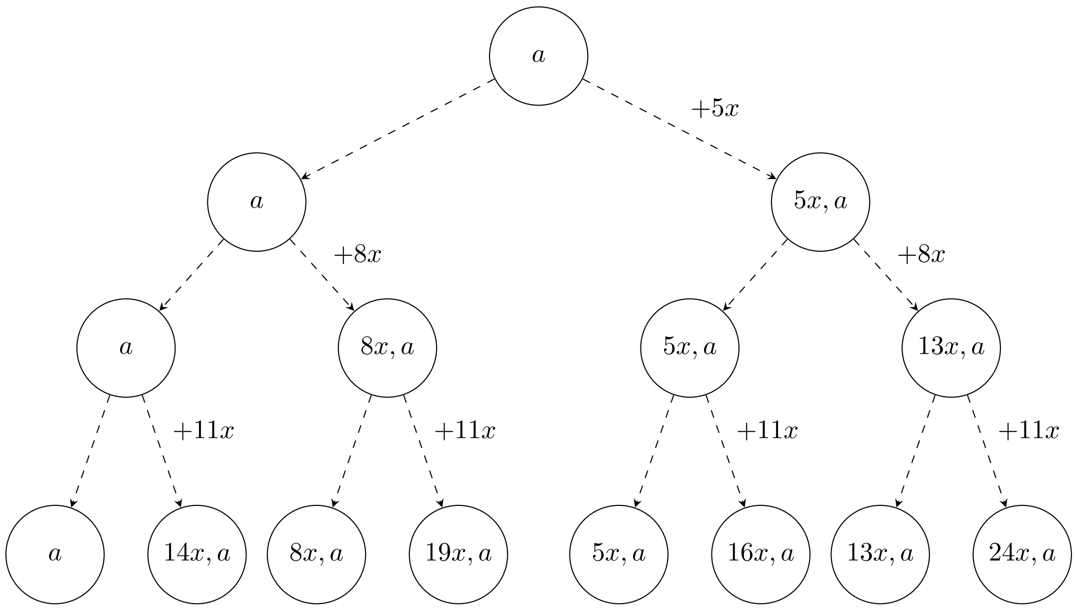

Figure 3.3: Generating an exponential computational space in linear time for a Kernel P system based solution to the Subset Sum problem

lowing: 1. division rules are restricted to a single application per compartment per step; and 2. since the newly created compartments inherit (a copy of) the multiset produced by their *parent* during the very same step, a clear precedence of traditional multiset rewriting and communication over division rules can be observed. Indeed, it is confirmed in [78]: "we may suppose that first the evolution rules of type (a) are used, changing the objects, and then the division is produced [...]"[1].

3.7 The membrane computing paradigm

The P system models examined in this chapter evince the diversity, the novelty and complexity infused in the membrane computing paradigm. The great variety of features and extensions is a testament to the inspirational quality exhibited by membrane systems, which gave rise to a remarkable creative momentum. One research direction which ensued can be characterised as *explorative* or *expansive*,

[1]Our remark underlines the qualitative distinction between typical multiset transition rules and division rules, which is evinced by the maximally parallel execution strategy. Stylistically, we find the interplay between compartments (context) and multisets (content, substance) organic and consistent with other P system innovations, although, from a very strict logical perspective it may seem unharmonious. Evidently, there is no question about the logical consistency of the model.

seeking to extend established models in innovative ways, introducing new primitives and elevated semantics. The aim was to generally *enhance* the expressive power of P systems, allowing for elegant solutions to specific problems/scenarios. Tissue and Population P systems as well as P systems with active membranes are eminent examples of this class.

The complementary tendency, of equal significance, was to identify a more *restrictive* set of primitives amidst newly emerged variants and establish their expressive power and complexity in isolation. This impulse sought to *refine* existing models and determine the functional significance of key elements more sharply. The study on the power of communication by symport/antiport rules [75] discussed earlier, as well as numerous investigations on the efficiency of Spiking Neural P systems and the universality results of the traditional P system model are noteworthy examples of this research direction.

Finally, a third category can be identified in this flux of developments. Models such as Neural P systems, Hyperdag P systems and Kernel P systems integrate an eclectic mix of elements and reconcile their often incongruous semantics in an effort to establish a unified and coherent modelling framework in the context of membrane computing. We have nominated these attempts as *syncretic*.

Whilst these diverging paths of development have engendered a diversity of related computation models, the originating concepts were also implicitly substantiated. We conclude this chapter by stating the most conspicuous features which typify the membrane computing paradigm:

1. A structured, distributed computational space;
2. Multisets of objects as
   (a) atomic terms in parallel transitions and
   (b) persistent data in delineated sectors of the computational space called compartments;
3. Intra-compartmental transition rules, generally known as (multiset) rewriting rules;
4. Inter-compartmental transition rules, often referred to as communication rules;
5. An execution strategy which orchestrates the application of rules.

Chapter 4

Formal Verification

Formal verification complements formal specification and development with the purpose of improving robustness and reliability of a model in a mathematically rigorous way. In contrast to other means of asserting program correctness and consistency, formal verification conveys a categorical response to an examination, by proving or disproving a correctness claim. The methodology generally requires:

1. an abstract mathematical model of the system,
2. an unambiguous representation of a set of properties to be verified,
3. a means of asserting whether the system complies with the stipulated properties/correctness claims.

The two most common approaches to formal verification are model checking and automated theorem proving. Each method presents both advantages and disadvantages which generally determine the suitability of one over the other in specific contexts. Whilst an ample and accurate comparison of the two approaches, documenting the criteria one should consider prior to selection, is itself a momentous undertaking, in this project we specifically address the formal verification of P systems by means of model checking.

The absence of an established semantic framework which parallel systems (such as membrane systems) can *non-reductively* map to is one primary consideration in this regard. A second aspect which motivates this research project is the ambivalence induced by various proceedings with similar objectives - these will be detailed later in this chapter. Is model checking a viable formal verification approach for P systems generally, or is it appropriate for particular variants or isolated cases only? What makes a P system model a suitable candidate for model checking? What are the limitations of model checking membrane systems, in contrast to other formalisms? These are some of the questions we commit to answering in this thesis.

4.1 Model checking

Model checking is a formal verification technique primarily used to prove the validity of finite-state reactive systems. In contrast to deductive systems which entail the systematic construction of mathematical proofs to assert program correctness, model checking is an exhaustive approach whereby the complete state space of the program is investigated. The technique was pioneered in the 1980s by Clarke & Emerson, and Queille & Sifakis [30,31,40,79] who showed that it can be feasible to check all possible computations of a concurrent program. The key insight in their approach is that both the model subject to verification and the correctness claim can be expressed as non-deterministic finite automata (NDFA). The verification problem is hence reduced to one in automata theory: a program P satisfies a property α if the intersection of the languages accepted by the corresponding automata, A_P and $A_{\neg\alpha}$, is empty. If a common string is found, then the computation accepting this input is a counter-example which invalidates the correctness claim.

A correctness claim is generally specified using *temporal logic*, a formalism developed by Amir Pnueli in 1977 [74]. Time related modal expressions such as 'the system will *eventually* halt' or 'the system *never* deadlocks' can be elegantly formulated using designated temporal logic operators, F (finally) and G (globally) respectively. Linear time temporal logic (LTL) [74] is one of the prominent formal frameworks in this respect. LTL consists of a finite set of *atomic propositions* $P = \{p_0, p_1, ..., p_n\}$, the logical operators $\neg$ and $\vee$, and the temporal operators X (next) and U (until). Formally, the set Φ of LTL formulae over P is inductively defined as:

- $\forall p \in P$, $p \in \Phi$;
- If α and β are formulae in Φ, then so are $\neg\alpha$, $\alpha \vee \beta$, $\mathrm{X}\alpha$ and $\alpha \ \mathrm{U} \ \beta$.

We can interpret the computation of a program as a (possibly infinite) sequence $P_0, P_1, ...$ of subsets from P, that is, a sequence of truth evaluations for atomic propositions in P. Such a sequence $M : \mathbb{N} \to 2^P$ *satisfies* an LTL formula α at instant i (relation denoted as $M, i \models \alpha$) as follows:

- $M, i \models p$, $p \in P$, iff $p \in M(i)$;
- $M, i \models \neg\alpha$, iff $M, i \not\models \alpha$;
- $M, i \models \alpha \vee \beta$, iff $M, i \models \alpha$ or $M, i \models \beta$;
- $M, i \models \mathrm{X}\alpha$, iff $M, i+1 \models \alpha$;

- $M, i \models \alpha \text{ U } \beta$, iff $\exists k \geq 1$, such that $M, k \models \beta$ and $\forall j, i \leq j < k$, $M, j \models \alpha$.

The additional logic operators are defined in relation to the primary elements of LTL:

- $\alpha \wedge \beta \equiv \neg(\neg\alpha \vee \neg\beta)$;
- $\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$;
- $\alpha \leftrightarrow \beta \equiv (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$;
- $true \equiv p \vee \neg p$, $p \in P$;
- $false \equiv \neg true$.

Furthermore, there are three derived temporal operators which are based on the 'until' (U) operator. Their validity is described as follows:

- $M, i \models \text{F } \alpha$ iff $\exists k$, $k \geq i$, such that $M, k \models \alpha$;
- $M, i \models \text{G } \alpha$ iff $\forall k$, $k \geq i$, $M, k \models \alpha$;
- $M, i \models \alpha \text{ R } \beta$ iff $M, i \models \text{G } \beta$ or $\exists k \geq 1$, such that $M, k \models \alpha$ and $\forall j, i \leq j \leq k$, $M, j \models \beta$;

More succinctly, the operators can be defined in relation to U:

- $\text{F } \alpha \equiv true \text{ U } \alpha$ (α eventually becomes true);
- $\text{G } \alpha \equiv \neg \text{ F } \neg\alpha \equiv false \text{ R } \alpha$ (α is always true).
- $\alpha \text{ R } \beta \equiv \neg(\neg\alpha \text{ U } \neg\beta)$ (β remains true until and including the point when α becomes true. β must hold indefinitely if α never becomes true);

For the purpose of verification, finite state reactive systems as well as (negated) temporal logic formulae are translated to Büchi automata [22]. A Büchi automaton is a non-deterministic finite state automaton which takes infinite words as input.

Formally, a Büchi automaton is a tuple:

$$A = (Q, \Sigma, \delta, I, F)$$

where:

Q is a finite set of states;

Σ is a finite set of symbols which represents the alphabet of A;

$\delta : Q \times \Sigma \to Q$ is the transition function of A;

$I \subseteq Q$ is the set of initial states of A;

$F \subseteq Q$ is a finite set of *final* (or sometimes called *good*) states;

A Büchi automaton accepts an input w if at least one of the infinitely occurring states over w is in F. The language recognized by A, denoted $L(A)$, is the set of all infinite words accepted by A. More generally, Büchi automata recognize the omega-regular languages - regular languages with infinite words.

The model checking problem is reduced to the construction of the product automaton of the two Büchi automata (corresponding to the reactive system and property expressed as a temporal logic formula) which identifies the intersection of their accepted languages. Specifically, a formula α is satisfied by a program P if *the intersection of the languages recognized by the Büchi automata corresponding to $\neg\alpha$ and P is empty.* Formally, this is stated as follows:

Let P be a program, α a formula and A_P and $A_{\neg\alpha}$ the constructed Büchi automata for P and $\neg\alpha$ respectively. Then all computations of P satisfy α if and only if $L(A_P) \cap L(A_{\neg\alpha}) = \emptyset$.

4.2 Model checker tools

One of the most fruitful advantages of model checking is the fact that it can be completely automated. The technique outlined in the preceding section has been implemented by numerous software tools to provide mainstream means for formal verification. The two most important aspects model checker tools focus on are:

- generating a system's state space efficiently and
- state storage strategies which also facilitate optimal retrieval for an overall efficient traversal of the state graph.

In order to achieve an efficient state space expansion, a model checker generally employs an 'on the fly' construction approach, such that all *reachable states* are generated progressively, *when* and *if* required in a particular verification instance ('run'). The model's state space construction and traversal is hence determined not solely by the problem size, but also by the type of properties whose validity is to be asserted. A 'finally' claim, for instance may be satisfied by a fractional subset of a system's state space; the 'on demand' approach is a remarkable enhancement in such cases.

The second topic of significance model checker tools are inherently concerned with arises from the fact that systems often exhibit recurrent states - this is always the case with reactive systems whose state transitions can only be modelled by Büchi automata (or other formalisms which allow infinite inputs). Thus, a model checker must first traverse the state graph constructed up to present and establish whether a state already exists, in which case a duplicate will not be added. It is implicit that a model's state space is a graph whose vertices correspond to *unique* states of the system. The *state search operation* is of primary importance and as such, considerable effort is invested in optimising the algorithms and data structures which store the state space (modelled as a directed graph) but also facilitate a speedy retrieval of state data. To this end, model checkers employ hash tables coupled with efficient hashing and compression algorithms. This is often supplemented by *partial order reduction* techniques which are essential to models of concurrent systems. Since a model checker reconciles parallel state transitions inherent to distributed systems by *interleaving atomic instructions*, it is frequently the case that not all independent paths are pertinent to the verification process. A partial order reduction strategy essentially prunes irrelevant paths resulted from process interleaving. Additionally, some model checkers also include BDD (binary decision diagram) based storage techniques to further optimise the execution of the verification procedure.

The methods utilised to improve and optimise the model checking approach also hint at the infamous drawback associated with the technique. Model checking is not scalable for distributed systems. The verification problem becomes intractable due to the exponential state expansion, relative to the number of processes of a system. This is a direct consequence of parallel transitions being reduced to interleaved instructions, which entails that all state combinations between the processes must be considered in the global state graph.

The intransigent *state explosion* problem may seem as contradictory to our earlier affirmation, regarding the prevalence and suitability of model checking for reactive, concurrent programs. In practice, formal verification by model checking is used to demonstrate the correctness of a methodology, the implementation of an algorithm or protocol, a segment of an application (i.e. a sub-routine, module, etc) and not *reducible instances* of a problem. This does not imply that model checking is generally applicable and permanently viable, however.

The state explosion problem can also be *avoided* by model construction when the system does not require a translation to a *traditional process model* (i.e. a model whose atomic instructions are interleaved). We anticipate, this is precisely the case with membrane systems.

4.3 The SPIN model checker

Developed by Gerard J. Holzmann in the 1990s, SPIN [8, 52] is a leading formal verification tool, widely used by both scientists and software engineers. Originally designed for verifying communications protocols, SPIN is particularly suited for modelling concurrent and distributed systems by means of interleaving atomic instructions. The tool can be used to validate the logical consistency of a specification, identify deadlocks, report race conditions and incompleteness.

A model checker requires an unambiguous representation of its input model, together with a set of correctness claims, generally expressed as temporal logic formulae. SPIN features a high level modelling language, called Promela (process meta language), which specializes in concise descriptions of concurrent processes and inter-process communication. A practical and highly convenient aspect of the tool is its complementary support for embedded C code as part of the model specifications. This confers outstanding flexibility in describing the behaviour of a system in its state transitions.

Correctness properties can be specified in SPIN as:

- process invariants using *assertions*;
- linear temporal logic (LTL) formulae;
- Büchi automata;
- general omega-regular properties in the syntax of SPIN *never claims.*

Whilst SPIN is a fully featured LTL model checker, it can also be used to verify more basic *safety* and *liveliness* properties which can be expressed using an alternative notation [6].

Other notable merits of SPIN include its support for multi-core computers, an ‘on the fly’ construction of the global state graph, support for mixed communication (both synchronous and asynchronous) between processes, efficient partial order reduction techniques for the verification of concurrent systems.

As a mature, established, open source software verification tool, SPIN is also the model checker of choice for this research project. Its performance but also its support for embedded C code which can effectively *hide* more complex, compound instructions and, generally support the implementation of ‘unconventional’ concepts such as maximal parallelism, are just two of the reasons which motivate our selection.

4.4 P systems and model checking

An initial investigation on the feasibility of model checking membrane systems [33] suggests that it is *theoretically* possible to verify P system models with *bounded multiplicities* on objects. Specifically, the paper focuses on the decidability of the model checking problem for traditional P systems. The experiments conducted (using two verification tools, SPIN and Omega) appear inconclusive however, indicating the method is impracticable for most properties: “Unfortunately, SPIN could not finish any run within one hour [...]”. A liveliness property was presented as an unexpected exception of this analysis. Indeed the concluding remarks addressed the necessity for improved modelling strategies: “more research is needed for both approximation methods to create a more efficient encoding” [33]. It is important to emphasize that each compartment was mapped to an individual Promela process and communication rules are implemented using *rendezvous message passing* between processes.

An interesting approach to LTL model checking in the context of membrane computing is presented in [14]. The paper demonstrates an executable algebraic specification of transition P systems for the Maude [32] rewriting engine. The focal point of this undertaking is the algorithmic implementation of the maximally parallel multiset transitions, inherent to membrane systems, in rewriting logic, executable by Maude. Whilst the question of formal verification is peripheral to this study (only two simple properties were examined), it was nevertheless shown that LTL model checking can be achieved in this setting, using an implementation of P system operational semantics in conjunction with Maude’s linear time temporal logic module, both expressed in a consistent rewriting logic notation. The tool automates the *parallel execution* of the two specifications, yielding a result to verification enquiry.

No performance metrics or more general remarks on the feasibility of this approach in an extended context can be derived from this investigation. It is notable, however, that this represents the first (and only, to the best of our knowledge) work which does not consider a translation of P systems to *process models*, a conventional requirement for model checking distributed systems.

A separate course of development [55, 56, 59] diligently pursued a general solution to model checking membrane systems using SPIN. The most remarkable aspect of the proposed methodology is the hybrid modelling technique, pertaining the system subject to verification and the correctness claims. Although the models investigated are supported by a single Promela process, the global reachability graph is abounded with irrelevant states (referred to as ‘intermediate’ states [56]), that is, states which correspond to atomic instruction interleavings, but are not part of the set of possible configurations of the P system. In order to *disregard* the superfluous states generated by the model checker, a boolean variable is included (in the state vector) which evaluates to true if the state is a genuine P system state and false otherwise. This synthetic distinction is then utilised in the LTL formulae to preserve the consistency of the verification process for a particular property. For instance, a formula G $a = 0$ may be invalidated if the intermediate steps are considered, however it is proved faithful to the computations of a P system if adjusted accordingly: G $a = 0 \vee \neg pSystemState$, where $pSystemState$ is the flag which differentiates between *intermediate* and *genuine* states. A complete translation strategy (pertaining all LTL temporal operators) is formally documented in [56]. A variety of properties have been verified across several (restricted) instances of a P system model[1], demonstrating the correctness of the approach on the one hand, and acknowledging its severely undermined scalability on the other hand. Moreover, the study also features a performance comparison (i.e. using the same models and properties) between SPIN and NuSMV [4], a symbolic model checker based on binary decision diagrams. The study indicates that SPIN is significantly more efficient for the purpose of LTL model checking, in the context of the two case studies.

Finally, in [38] we presented an extrapolation of this technique, applicable to Kernel P systems with multiple compartments. Despite the fact enhancements were also introduced in an attempt to exclude (*hide*) ancillary computational paths from the global state graph generated by the model checker, we have concluded that only an approximate formal analysis can be conducted on such models, using SPIN's BITSTATE mode (also referred to as supertrace interpretation).

The following observations are apposite to our proposed modelling approach, described in the following section:

1. All summarised undertakings which target a dedicated model checker tool (i.e. excluding the Maude rewriting engine) promote a mapping to a *standard process model*, whereby each individual compartment is associated a separate process, whose atomic instructions are interleaved to account for the parallel behaviour of the model.

2. With the exception of the latter investigation (targeting Kernel P systems), all referenced case studies consider traditional (cell-like) P systems with one or two compartments and are generally restricted instances. This is to say that *there is no integrated approach which pertains the membrane computing paradigm in its computational model diversity.*

[1]Since a translated P system model is decorated with auxiliary variables necessary to express certain LTL formulae, it must always be bounded to satisfy the requirement of having a finite state space.

4.5 Sequential models for parallel systems

Membrane systems are often referred to as *unconventional* computational models; the membrane computing paradigm the models substantiate is widely regarded as singular. Since the term 'unconventional' suggests more than candid novelty, it is natural to ask what are the key determinants for this status? Is this classification justified or insubstantial? What is the *convention* P systems diverge from?

Whilst membrane systems are indeed an irregular convergence of concepts with direct biological correspondences (membrane, cell, tissue, symport-antiport, dissolution, etc), into an abstract formal model that is computationally universal, we believe a P system's originality is identified in the vision of parallelism it implements.

The *conventional models* for distributed systems were conceived in response to a practical concern, namely, how could the *asynchronous* dynamics of *distinct* entities be modelled by sequential computation. The answer was, of course, to *abstract away time.* The primary objective of such endeavours was to reconcile an agnostic view of parallelism algebraically. If no time related inferences can be made (i.e. it is assumed that *it cannot be known how two distinct transitions execute in time, relative to each other*), a mathematical model must be capable of expressing all possible computations of an aggregate system which subsumes the behaviour of individual *processes.* This modelling approach became generally known as *process calculus* (or process algebra) and includes the CSP (Communicating Sequential Processes) [51], CCS (Calculus of Communicating Systems) [64], ACP (Algebra of Communicating Processes) [18], and π-calculus [65] amongst other formalisms. Parallelism is not directly affirmed in process calculi[1], but rather acknowledged, inferred and *reconciled operationally.* The parallel composition operator epitomises the algebraically *constructive* means of expressing concurrency. The result of a formula $P \mid\mid Q$ (using CSP notation [51]), which denotes the parallel composition of two processes P and Q, is another process R whose specification must account for *all possible ways* the composite can advance. This is captured by the following equational statement, applicable when the alphabets of P and Q are disjoint:

$$(a \to P) \mid\mid (b \to Q) = (a \to (P \mid\mid (b \to Q)) \mid b \to ((a \to P) \mid\mid Q))$$

The two processes must *synchronise* on events which are common to their alphabets. Thus, for an event $c \in \alpha P$ and $c \in \alpha Q$ (is in the alphabet of P and

[1]A remarkable aspect which echoes the agnostic perspective assumed, is the minimalist title of each formalism: a plurality of independent sequential processes is explicit, however, parallelism is not addressed as a generic concept.

Q, respectively), then the $P \mid\mid Q$ can only advance if the two processes engage in event c simultaneously:

$$(c \rightarrow P) \mid\mid (c \rightarrow Q) = c \rightarrow (P \mid\mid Q)$$

The semantics of this operation can also be described in terms of the *traces* of R, relative to the operands P and Q. A CSP trace of the behaviour of a process is a finite sequence of symbols which records the events the process has engaged in, up to some moment in time [51]. The traces of $P \mid\mid Q$ are defined as:

$$traces(P \mid\mid Q) = \{t \mid (t \restriction \alpha P) \in traces(P) \wedge (t \restriction \alpha Q) \in traces(Q) \wedge t \in (\alpha P \cup \alpha Q)^*\}$$

where $(t \restriction \alpha P)$ denotes a trace t restricted to the alphabet of P.

Whilst an algebraic formal representation of distributed system dynamics is essential to the formal reasoning of concurrent behaviour, the intrinsic cost of this reduction is the *non-deterministic choice* introduced in the resulting process for each pair of asynchronous transitions. It is this projection to non-deterministic branching which inevitably leads to an exponentiation of the number of states for a parallel composite process.

To illustrate this consequence, we consider three simple, deterministic[1] CSP processes with disjoint alphabets:

$$P_0 = a_1 \rightarrow a_2 \rightarrow STOP.$$

$$P_1 = b_1 \rightarrow STOP.$$

$$P_2 = c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow STOP.$$

The processes can be represented as state transition systems, as depicted in fig 4.1. The state space of the process $R = P_0 \mid\mid P_1 \mid\mid P_2$ is defined by the set of state tuples $S_R = \{(s_i, s_j, s_k) \mid s_i \in S_{P_0},\ s_j \in S_{P_1},\ s_k \in S_{P_2}\}$, where S_{P_n} denotes the set of states of P_n. Thus, the number of states of the composite process R is equal to the product of cardinalities of its constituents: $card(S_{P_0}) \times card(S_{P_1}) \times card(S_{P_2}) = 24$.

Membrane systems are situated in direct antithesis to the *reductive* modelling approach of process calculus. Rather than reconciling concurrent dynamics to sequential, algebraic constructs, P systems *elevate* all individual object transformations to a single, atomic transition. In this respect, a P system's maximal

[1]The purpose of this example is to illustrate the combinatorial expansion of states which arises due to the reduction of parallel, asynchronous transitions to non-deterministic choice in a sequential process. The non-deterministic choice in a individual process simply increases the number of states one must consider in the combinatorial expansion.

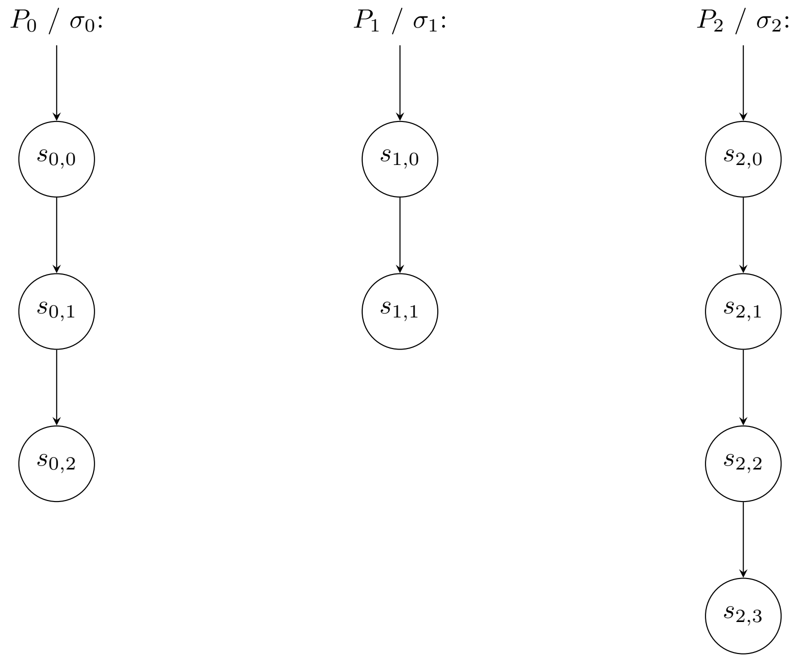

Figure 4.1: State transition systems for three processes.

parallelism is, as its name implies, *expansive*: a compartment with three a objects will execute a rule $a \longrightarrow a^2, b$ three times, in a single, indivisible unit of time; the same compartment will apply the rule nine times in the next computational step. This execution strategy is extrapolated to multiple compartments which generally constitute a membrane structure. Each atomic instruction, regardless of the locus of execution, *participates organically to a P system's unitary transition.* There is always a single progression that is exercised and that is the sequence of configurations of the P system - the single identified sequence overarches the model in its entirety; there is no sequential behaviour attributed to individual compartments (functional units).

This distinctive, 'unconventional' parallelism has direct implications to the system's state space construction and evaluation. We consider a simple, deterministic P system, comprising of three compartments, whose individual transitions are precisely reflected by the state transition systems in fig 4.1:

$$\Pi_{eg} = (O, H, \mu, \omega_{\sigma_0}, \omega_{\sigma_1}, \omega_{\sigma_2}, R_{\sigma_0}, R_{\sigma_1}, R_{\sigma_2}, i_0)$$

where:

$O = \{a, b, c, d\}$;

$H = \{\sigma_0, \sigma_1, \sigma_2\}$;

$\mu = [[[\ ]_2]_1]_0;$

$\omega_{\sigma_0} = a^2$, $\omega_{\sigma_1} = a^3$, $\omega_{\sigma_2} = a;$

$R_{\sigma_0} = \{a \longrightarrow b;\ b \longrightarrow c\}$

$R_{\sigma_1} = \{a \longrightarrow b\}$

$R_{\sigma_2} = \{a \longrightarrow b;\ b \longrightarrow c; c \longrightarrow d\}$

$i_0 = e.$

We note that Π_{eg} halts after precisely three steps with configuration $C_3 = \{[[[\ ]_3]_2]_1, (c^2, b^3, d)\}$. The state space of Π_{eg} is captured by the expression: $S_{\Pi_{eg}} = \{(s_{i,x}, s_{j,x}, s_{k,x}) \mid s_{i,x} \in S_{\sigma_0,x}, s_{j,x} \in S_{\sigma_1,x}, s_{k,x} \in S_{\sigma_2,x}, 0 \leq x \leq 3\}$, where $S_{\sigma_t,x}$ is the set of states corresponding to a compartment σ_t *at computational step x.* The number of states the P system (as a whole) can situate in is exactly four; this is generally given by the compartment with the maximum number of states: $max(card(S_{\sigma_0}), card(S_{\sigma_1}), card(S_{\sigma_2}))$, where $card(S_{\sigma_n})$ denotes the cardinality of the set of states corresponding to σ_n.

We clarify our decision to exclude non-determinism from the above examples and concluding remarks. Non-deterministic choice introduces a combinatorial state expansion in a sequential model, *regardless* of the formalism utilised and the concurrent behaviour of the system. The argument we have forwarded aims to delineate the two general views on concurrency more sharply and evince a very practical consequence of a non-reductive approach to parallelism (i.e. one whereby independent transitions are not reduced to non-deterministic strands). Non-deterministic choice, as a modelling concept, is hence extraneous to our hypothesis.

The key implications with respect to formal verification are the following:

Proposition 1. *Although membrane systems are inherently distributed models, their projection to traditional process models (as advanced by process calculi) is inadequate. The interpretations of parallelism promoted by the two formalisms stand in direct opposition.*

Proposition 2. *Model checking is a particularly suitable formal verification technique for P systems due to the non-exponential state expansion, relative to the number of compartments, the model exhibits.*

The exclusion of instruction interleaving between the compartments of a P system enables us to avoid the state space explosion problem for deterministic models. Evidently, P systems featuring non-deterministic selection for multiset transition rules will relentlessly generate an exponential state space. Our thesis simply deprecates the modelling of concurrent dynamics by means of non-deterministic transitions (for membrane systems), which equate to interleaved instructions when considering a model checker tool that operates sequentially.

We therefore introduce a new sequential modelling strategy for P systems, targeting SPIN's Promela specification. The approach diverges from conventional process modelling and reflects the outcome of this examination. We outline its principles below; a more detailed description of the mapping to Promela data types and procedures is presented and exemplified in chapter 6.

- A single SPIN process supports the execution of a P system and implicitly, all compartments it consists of;
- The inter-compartmental exchange of objects is represented as a pair of update instructions, similar to rewriting rules, and not by native inter-process communication routines in Promela;
- Each set of rules associated to a P system compartment is evaluated and applied exhaustively *in succession*. This is because the order in which compartments are selected for rule application is irrelevant - the same P system state will be reached;
- All atomic instructions which constitute a complex P system rule are *concealed*; All applicable P system rules which collectively define a state transition should be treated by SPIN as a single, indivisible (i.e. atomic) functional unit.

Parallel transformation is not implied in membrane systems but rather it is *affirmed* as a transition sui generis. It is characterised as maximal or expansive because all possible mutations can occur independently, as part of the same, singular transition. We have underlined the significance of this consideration for the purpose of model checking. The sequential modelling approach set forth in this thesis is a decisive element for the feasibility of model checking P systems.

Finally, we wish to re-affirm that the distinction we have forwarded in this chapter, between the two interpretations of parallelism is *not* one of value nor *functional efficacy*. We do not discuss the functional capacity to model concurrent dynamics nor the computational equivalence between process calculi and P systems. We have concluded that process models are *inadequate* not on the basis of deficiency, but due to the conceptual disparity with membrane computing, on the notion of parallel transitions. The two formalisms are examined as antithetic modelling paradigms, one which aims to preserve the sequential aspect of computation for an individual process in a context where multiple processes co-exist and may interact; and the other promoting an organic, unitary transition across all mutable elements of a system. Our assertion is, of course, circumscribed in the objective of this research project, that is the formal verification of P systems by model checking, and its relevance is described therein. One need not consider a CSP process as identified by its traces in all circumstances, however, when the goal is the exhaustive investigation of a program's state-space, this is an essential requirement.

We also note that by 'inadequate' it is not implied that process calculi are incapable of representing P system transitions[1], nor does this suggest a qualitative inferiority of CSP (and other other process calculi) in their representation of the (present) notion of parallelism[2] computationally. Rather, it expresses a profound incongruity between the individual transitions of processes in process algebra and the atomic transitions of P system compartments: a direct mapping between membranes with associated multiset rules and processes with actions is erroneous. We concede, this argument is not scientifically formulated (since it was not scientifically derived), however its consequences (for the purpose of model checking) can be ascertained within a scientific framework, which is an objective of this research project.

[1]Whilst traditional process algebra do not have provisions for initiating an arbitrary number of processes during one transition, there are indeed models (e.g. ambient calculus [24]) apt for simulating the maximally parallel execution strategy of P systems.

[2]We do not present our views on parallel dynamics as a generic concept in this thesis. We acknowledge, however, process calculi, P systems, Petri nets and other formalisms mentioned in chapter 2 as models which articulate a perspective on this concept, as perceived scientifically.

Chapter 5

Elementary P systems

An elementary P system (EPS in short) is a computational model which aims to capture the membrane computing paradigm in its nuclear essence and subsumes the feature diversity and potency of its kindred models into a single formalism.

Whilst research in membrane computing has already advanced a series of models which aggregate an eclectic mix of primitives, common to select classes of P systems (earlier, in chapter 3, we have called these developments *syncretic*), the variants defined in this respect instilled complex semantics in an attempt to reconcile the *irreducible functional diversity* inherited. As consistent formal models, P system variants such as Neural P systems, Hyperdag P systems or Kernel P systems are convenient modelling frameworks with outstanding expressive power in specific contexts. The coherent linear mapping to atomic elements of a mathematical model is not gratuitous when the abstract model must cater for complex ontologies and structures. The intricate semantics derived in such models are particularly detrimental when formal verification is a set primary objective.

Elementary P systems represent an effort to distil the P system functional vocabulary whilst preserving fundamental concepts which constitute the bedrock of membrane computing. EPS models were designed with an aspiration to minimise the set of primitives and apply simple, coherent semantics. Moreover, elementary P systems effectively integrate distinctive features of the most prominent variants (such as promoters/inhibitors, membrane creation/division, dissolution, targeted, replicated communication etc), allowing for effortless, intuitive translations. Not all concepts, however, are explicitly present - that is, they are not expressed through dedicated primitives, but rather derived from the core components and their operational capacity. Generally, expressive power is retained not with the provision of various specialised, irreducible abstractions, but rather by augmenting the significance (semantics) of the existing primitives in specific configurations.

In this chapter we present elementary P systems extensively, as a distributed and parallel computational model rooted in the context of membrane computing.

5.1 Definition

An elementary P system is a three tuple:

$$e\Pi = (O, C, R)$$

where

- O is a finite set of symbols denoting objects - the system's alphabet;
- C is finite set of non-empty *compartments* and represents the model's initial configuration;
- R is a finite set of multiset transition rules.

A compartment denotes a multiset data structure which persists across the system states. The concept diverges from its original significance found in traditional P system models where a compartment is itself a discrete entity which *houses* a multiset and has a unique label. An EPS compartment is by contrast incorporeal, directly identified by its multiset and it is not uniquely addressable. The reflexive binding and unity between a compartment and its multiset in elementary P systems makes this distinction diaphanous and as such, a reference to a compartment $c \in C$ is generally regarded as a reference to c's associated multiset. Examples: (a^3, b^2), (x, y, z^5), (t^{10}).

Since compartments are the sole volatile components of an EPS, they collectively constitute the configuration of the system at a particular computational step. Thus, a configuration $M_i^{e\Pi}$ is equal to the set C_i, the set of compartments C at step i. An $e\Pi$ computation is defined as a sequence of configurations $M_1^{e\Pi} \Longrightarrow M_2^{e\Pi} \Longrightarrow ... \Longrightarrow M_h^{e\Pi}$.

A multiset transition rule specific to elementary P systems can be generically represented as *scope* : $lhs \longrightarrow rhs$, where *scope* designates the set of compartments the rule is applicable to; the lhs (left hand side) term denotes a non-empty multiset over O which is to be 'consumed' from the compartment the rule is applied in; the rhs (right hand side) is a multiset complex which denotes the outcome of this transition relative to both the compartment the rule is applied *in* and the P system as a whole.

More formally, a *scope* is a pair $\langle p, !q \rangle$, where p, q are multisets over O. We say a compartment c *is congruent with* a scope s ($c \equiv s$) if and only if:

1. $\forall x \in p, |p|_x \leq |c|_x$ and

2. $\forall x \in q, |q|_x > |c|_x$.

where $|p|_x$, $|c|_x$ and $|q|_x$ denote the number of occurrences (the multiplicity) of symbol x in multisets p, c and q respectively.

Conversely, $c \not\equiv s$ if any of the aforementioned conditions are not satisfied. We exemplify scopes and scope congruence: $(a, b^2, c^5) \equiv \langle a \rangle$; $(a, b^2, c^5) \equiv \langle a, b, c \rangle$; $(a, b^2, c^5) \equiv \langle b^2, c^5, !d \rangle$; $(a, b^2, c^5) \not\equiv \langle !a \rangle$; $(a, b^2, c^5) \not\equiv \langle a, b^2, !c^3 \rangle$; $(a, b^2, c^5) \not\equiv \langle !b^3, c^5 \rangle$.

A scope is considered to be empty if both p and q are empty multisets ($p = \lambda$ and $q = \lambda$). An empty scope is never explicitly stated but rather the absence of a scope in a construct can be envisaged mathematically as $\langle \lambda, !\lambda \rangle$ (that is to say, the concept is ancillary and not strictly required, however it underpins the logical consistency of the model). An empty scope matches *all non-empty* compartments:

$$\forall c \in C, c \equiv \langle \lambda, !\lambda \rangle \Leftrightarrow |c| > 0$$

The *rhs* term can be formally described as a union of *target products*:

$$rhs = \bigcup m_{target}$$

where each m_{target} is a multiset destined to a particular compartment or set of compartments. The *target* denominates the recipients of the multiset outcome m, either directly or indirectly with the following indicators (for clarity and conciseness we assume an arbitrary rule $r \in R$ executing in an arbitrary compartment $c \in C$):

- *self* - implies the multiset m is to be contained by the same compartment c the rule is executed in;

- *all* - signifies that all non-empty compartments in C, except for c (self), will receive a copy of m;

- *new* - indicates that multiset m is sent to a *new*, *empty* compartment which is not yet part of C. Such a *latent* compartment is available to each existing $c \in C$, at each computational step, however it is only instantiated if at least one rule with target *new* is applied;

- *scope* - a pair of multisets $\langle p, !q \rangle$ which serves as a selection criterion across the set of compartments C: a copy of m is emitted to all compartments $c \in C$ congruent with scope $\langle p, !q \rangle$. If *scope* $= \langle \lambda, !\lambda \rangle$ - the empty scope, then the indicator is semantically equivalent to *all*.

We illustrate these definitions with the following examples:

1. $a \longrightarrow (b^2, e)_{self}$ - a multiset transition rule which takes one a and outputs two b and one e objects in the compartment the rule was applied to. For convenience, the rule can also be written as: $a \longrightarrow b^2, e$, thus omitting the target indicator; We also note the absence of a preceding scope which equates to global availability of this rule for an EPS $e\Pi$;

2. $\langle m, n^2 \rangle :\ a, b^2, e \longrightarrow x_{self}, y^2_{all}, z_{k^2,!t}$ - the rule is applicable in compartments with at least one m and two n objects, expending one a, two bs and one e and producing an x in the current compartment, broadcasting two ys to all other compartments in C and lastly, sending one z to compartments with at least two k objects and no ts;

3. $\langle q, !q^2 \rangle :\ x \longrightarrow y_{new}, w_m, w^2_{!m}$ is applicable in compartments with exactly one q; the rule consumes one x and sends one y to a latent 'new' compartment (an in-depth treatment of this concept will follow), a w object to compartments which contain a minimum of one m and two ws to compartments with no ms, respectively.

It is apparent that an elementary P system transition concentrates the dynamics intrinsic to most P system models, formally expressed as multiset rewriting, communication and membrane division/creation rules. For instance, a rule $a \longrightarrow b, c^2$ can be clearly regarded as a multiset rewriting rule whilst $x \longrightarrow y_p$ resembles targeted communication. As evident as this correspondence may be, the disparities are also prominent: a scope is not semantically equivalent to the notion of *promoters and inhibitors* (featured by P systems with permitting/forbidding contexts), nor is this an ancillary conditional, similar to the *states* of Hyperdag P systems; communication is not solitary - objects are propagated radiantly to all or a restricted subset of compartments in the system; new membranes are not created in the traditional sense where they inherit the content of the parent compartment, but rather additional compartments are *assumed* in a latent phase and instantiated as objects are channelled to it.

All rules declared for an EPS model are executed in a maximally parallel manner. The emblematic execution strategy, in conjunction with multiset transition rules, is considered the foremost novelty of membrane computing and hence, one of the *elementary* constituents of this model. Since all rules operate on multisets of objects, no additional stipulations are required to clarify the model semantics when compartment expansion or reduction is considered.

Whilst the $a \longrightarrow a'_{new}$ may be reminiscent of the membrane division rules characteristic to P systems with active membranes, it is important to emphasize that a transition to the *new* compartment participates to the maximal parallel rule execution similarly to any other contending rule. It is not an exceptional mutation exerted by a higher entity in the system (a membrane division rule is considered a structure changing rule and is applied on the compartment itself; consequently a membrane division rule may execute once and only once per step) and does not fracture the homogeneity of the maximal parallel execution strategy. Objects are transferred to the *new* compartment analogously to multiset communication. The instantiation of a quiescent compartment is implicit and occurs if at least one object has been inserted within. In contrast to P systems with active membranes, the content of the parent compartment is not copied into the newly created membrane.

Elementary P systems do not feature membrane dissolution rules. Compartments may, however, be invalidated and cease to be elements of the system's set C. This occurs when the respective compartment is devoid of content, that is, equal to the empty multiset λ. In fact C by definition cannot contain empty compartments, (λ). In EPS we prefer to say a compartment is *detached* from the system. We recall that membrane dissolution is a very specific concept in the traditional P system model: the execution of a dissolution rule effectively transfers a compartment's multiset content to its parent. When a compartment is detached from $e\Pi$ this simply signifies that it is no longer addressable and therefore vacuous. On the one hand there are no rules which can execute inside, since all rules are *assigned* to compartments by scope congruence at each step, on the other hand there are no communication rules which can deliver objects to this compartment, again because there is no scope which identifies (λ). Hence, the functional relevance of the multiset is doubly amplified in elementary P systems:

1. The presence of a multiset in a quiescent compartment achieves its instantiation and complementary

2. the absence of a multiset of objects in a compartment translates to its detachment from $e\Pi$.

We remark that C can never become empty $(\emptyset)$ - this is a corollary of the types of rules employed and their format:

1. Objects can be reduced by multiset rewriting but never completely depleted since

2. the right hand side of a rewriting rule must be a non-empty multiset;

3. An elementary P system with a singular compartment cannot execute any communication rules.

Furthermore, C must initially consist of at least one compartment. Having $C = \emptyset$ has no meaning.

One of the primary objectives of elementary P systems is to provide a sufficiently abstract formal model which may accommodate the feature diversity exhibited by the membrane computing paradigm. To this end, the exclusion of an explicit membrane structure from an EPS is perhaps the most remarkable, if unexpected, commitment. Whilst incorporating a definite structure of compartments offers conciseness and clarity of expression for communication rules in particular, the coordination (*out*, *here*, *in*, *siblings*, *parent*, *child*) is also intuitive and allows the formulation of powerful semantics (for instance, division and dissolution rules refer to the *parent* compartment to include or, respectively exclude to, entire multisets of objects). A fixed structure, explicitly stated by a P system constituent (generally denoted by μ), may, however, reduce the modelling scope not just because of a potential incompatibility in the distribution of compartments, but more importantly, due to the specificity of the rules employed. This is indeed detrimental to our goal of defining an abstract P system generic type.

The resolution adopted in elementary P systems was to freely admit any dynamic structure of compartments, *inferred* by scoped communication rules. Since scope congruence is a predicate applied on multisets of objects, no additional elements (such as arcs/channels with states, labels or nesting configuration) are required to guide the transfer of objects between compartments. The complexity of the system is effectively reduced by allowing existing, simple primitives to express a membrane structure by construction. On the one hand, EPS compartments are indeterminate - rules are dynamically allocated each step based on scope congruence and hence, conditioned by the multiset content at that step; and on the other hand, object transfer is channelled using the same congruence principle on compartments - the constraints imposed by a scope in the right hand side component of a rule give shape to a possibly dynamic structure which can be deemed as implicit to $e\Pi$. In chapter 7 we demonstrate how elementary P systems can model communication in a directed acyclic graph (DAG) structure.

An EPS halts when no further rules can be applied. The output of a halting computation is represented by the system's final configuration $M_h^{e\Pi}$. Whilst more specific values can be designated as a calculation result within this configuration, the model does not encompass any markers or filters to achieve this reduction. These can be formulated in various ways, independently. For example, one may consider an algorithm whose outcome is the number of c objects across compartments whose content does not include *root* or *leaf* objects. This could be expressed as $|\langle !root, leaf\rangle|_c$, utilising the concept of scope to identify a restricted set of compartments in C.

We next examine the functional correspondences between EPS and the principal P system models surveyed in chapter 3.

5.2 Elementary P systems and membrane computing

Whilst a morphological comparison between elementary P systems and the fundamentals of the membrane computing paradigm is instructive and generally implies a functional congruity, in this section we aim to define these correspondences more clearly, determine the less apparent limitations of EPS primitives and illustrate translation patterns between established P system variants (chapter 3) and EPS.

5.2.1 Membrane structure

The membrane structure is of key significance for P systems which feature communication, membrane dissolution or division rules. Relational data about the compartments of a model is utilised to effect a targeted multiset transfer. In a hierarchical structure of compartments (traditional P system models, P systems with active membranes, Hyperdag P systems), objects can be passed between child, parent and sibling compartments, whereas a graph-like membrane structure (Tissue, Neural, Spiking Neural, Population P systems) restricts the exchange of objects to linked compartments only (i.e. compartments which share a channel or edge). In elementary P systems, communication is freely admitted between any two compartments, using *scope addressing*. In order to designate one or more recipients, a scope element must accompany the multiset which is to be sent. The recipients are determined based on scope congruence during each computational step, facilitating the design of a highly general and dynamic compartment structure which is inferred by scope addressing in communication rules. Thus, in order to emit objects to a particular compartment σ_i, the multiset associated to σ_i (i.e. its content) must uniquely identify σ_i in the context of C, the set of compartments of an EPS $e\Pi$. A basic approach to this end is to consider a unique identifier (object in the set O of $e\Pi$) for each compartment σ_i of the model. This is of course limited to EPS models which do not exhibit membrane instantiation - the P system's alphabet is a finite set of symbols and it is pre-established. Alternatively, one may utilise the multiplicity of a particular object as an identifier and rules with scopes of the form $\langle a^x, !a^{x+1}\rangle$ to refer to a compartment with precisely x number of a objects. On this basis, multiset communication via an individual (directed) channel (i, j) can be represented by a rule $\langle a^x, !a^{x+1}\rangle : m \longrightarrow m'_{\langle a^y, !a^{y+1}\rangle}$, where multisets a^x and a^y encode identifiers i and j respectively. Since membrane instantiation is a concurrent operation, it is impossible to retain uniquely identified compartments indefinitely, using object multiplicities. Uniqueness can, however be guaranteed for a restricted number of instantiated compartments: consider a rule $\langle a\rangle : a \longrightarrow a^2_{new}$ executing in a compartment (a^{10}) - such a model will expand exponentially for ten computational steps, yielding a unique number of as in each instantiated compartment. Whilst limited, the technique does allow for single compartment referencing and channelled communication. This constitutes an advantage over P systems with active membranes where each newly create compartment must bear a label from a finite set H, which entails that unique compartment identifiers in P system with active membranes require an exponential increase in the cardinality of H and division rules applying the labels.

In a hierarchical structure, communication rules generally target parent or child nodes. The parent-child relationship can be encoded by a pair of objects (p_i, cp_i), such that each parent compartment σ_i in C is associated the object p_i and every child of σ_i will consist of an object cp_i. In this setting, the child compartment may transfer objects to its parent using rules of the form $\langle cp_i \rangle : a \longrightarrow a_{\langle p_i \rangle}$, whilst the converse can be achieved with $\langle p_i \rangle : a \longrightarrow a_{\langle cp_i \rangle}$. Moreover, communication between sibling compartments will employ the same cp_i object as a target indicator: $\langle cp_i \rangle : a \longrightarrow a_{\langle cp_i \rangle}$. Chapter 7 demonstrates this strategy in a dedicated case study and examines variations which may be advantageous in specific circumstances. We underline the contrast between EPS and traditional P system semantics for communication rules with multiple recipients: an EPS will replicate the object to all scope congruent compartments, whereas a recipient is non-deterministically chosen in traditional P system models. We will address this incongruity later in this section.

Since compartments are directly identified by their associated multiset, any membrane structure that is based on these identifiers is implicitly mutable. Importantly, structural mutation is attained by multiset rewriting and communication and not via specialised rules, such as link (channel) creation/destruction. New compartments are assumed 'instantiated' when objects are emitted using the *new* target indicator, in the same maximally parallel manner and not via membrane creation/division rules with distinctive semantics. It is sufficient to add or remove a objects from a compartment σ_i to prohibit the execution of a rule $\langle a^x, !a^{x+1} \rangle : m \longrightarrow m'_{\langle a^y, !a^{y+1} \rangle}$. Likewise, applying $\langle p_i \rangle : p_i \longrightarrow p'_i$ will invalidate the parent-child relationship encoded with the pair (p_i, cp_i). The relation can be later re-established with a rule $\langle p'_i \rangle : p'_i \longrightarrow p_i$[1].

[1]Since p_i and p'_i are considered unique compartment identifiers in the context of an EPS, the scopes $\langle p_i \rangle$ and $\langle p'_i \rangle$ prefixing these rules are in fact redundant. Nevertheless, the presence of a scope adds clarity and emphasises that execution of the respective rule is bound to a particular compartment.

5.2.2 Multiset rewriting rules

The multiset rewriting rules of elementary P systems are semantically equivalent to rewriting rules featured by traditional P system models. Formally, these are distinguished in a broader context with the $self$ marker, however this is conventionally omitted when contextual clarity is not compromised. Generally, a P system rewriting rule $u \longrightarrow v$, with u, v multisets of objects over the system's alphabet, has an identical representation in EPS and is applicable under the same conditions (i.e. the compartment must comprise a multiset w, such that $u \subseteq w$).

5.2.3 Multiset communication rules

As described earlier, EPS communication rules employ scope addressing in order to channel a multiset to one or more recipients. We also note that if there are no scope congruent compartments for at least one targeted multiset $m_{\langle s \rangle}$ which comprises the right hand side of a communication rule, the rule is inapplicable. This restriction holds for all P system variants with communication or rewrite-communication rules, excepting Hyperdag P systems. In order to simulate the exceptional behaviour of the latter model, one may fracture a communication rule with n segments $a \longrightarrow a'_{i\langle s_i \rangle}$, $1 \leq i \leq n$, into n individual rules with a distinct left hand side object to ensure each rule executes the correct number of times: $x_i \longrightarrow a'_{i\langle s_i \rangle}$, where x_i is a multiset with a single distinct object. A rule $a \longrightarrow x_1, x_2, ..., x_n$ must supplement this set of separate instructions and it must always execute in a dedicated step, preceding the application of the aforementioned n rules. Furthermore, a procedure is also required to re-generate y copies of the multiset a, where $y = min(x_1, x_2, ..., x_n)$, that is the minimum left hand side remainder of the n rule execution; all remaining $x_1, x_2, ..., x_3$ objects must also be disposed of during the following step. Thus, a minimum of two auxiliary computational steps are necessary to simulate this behaviour, together with n objects for each targeted multiset in each communication rule. The procedure is of course cumbersome, however, in concrete scenarios such semantics could be implemented using a mixture of features promoted by the EPS formalism. For instance, in order to isolate rules and dismiss the necessity for n distinct objects, we can distribute the n rules to n compartments[1]. Since elementary P system communication is expansive (i.e. a multiset is replicated to all scope congruent compartments), the n compartments can be simultaneously *signalled* to execute the independent communication rule associated.

Whilst object communication by replication is sufficient to represent the $\beta = repl$ transfer mode of Hyperdag and Neural P systems, its semantics is incompat-

[1] In order to associate each rule to a compartment, identifiers based on object multiplicity can be utilised, rather than distinct objects.

ible with the more traditional non-deterministic recipient selection, featured by the majority of the variants. In elementary P systems, non-determinism can only be expressed using two or more rules which expend at least one common object. More formally, $u \longrightarrow v$ and $u' \longrightarrow v'$, with u, v multisets over the alphabet O, are subject to non-deterministic selection if and only if:

1. Both rules are associated to and applicable in the same compartment $\sigma_i \in C$ and
2. $u \cap u' \neq \emptyset$.

Consequently, non-deterministic target selection must be encoded by means of individual communication rules with the same left hand side multiset. A P system rule $u \longrightarrow u_{in}$ which transfers multiset u to a random child compartment requires in EPS that all child compartments are uniquely identified such that references to these can be drawn. For each child compartment σ_i, a multiset a^i will serve as an identifier and n mutually exclusive rules $u \longrightarrow u_{\langle a^i, !a^{i+1} \rangle}$, $1 \leq i \leq n$ associated to the parent of σ_i will emit multiset u to a non-deterministically chosen σ_i. Employing the same technique, one may also prompt non-deterministic communication on groups of compartments in EPS. It is sufficient to have multiple child compartments congruent to the same scope which may resemble a range of multiplicities: $u \longrightarrow u_{\langle a^1, !a^3 \rangle}$, $u \longrightarrow u_{\langle a^3, !a^7 \rangle}$. The two rules ensure that either u is transferred to *all* compartments with less than three as, or it is sent to all compartments with at least three but no more than 6 a objects. This can not be easily achieved with other P system variants.

Antiport rules constitute another class of specialised instructions whose potential was evinced in several P system variants. Antiport communication can be interpreted as a synchronised exchange of objects between two compartments: (a_{in}, b_{out}) will execute in a maximally parallel context as long as there is an a object to extract 'in' (from a parent membrane) and a b object to send 'out'; that is, the rule will execute k times, where k is the minimum of the multiplicities of a and b in the respective compartments. In elementary P systems, such a rule can be specified as a triple:

1. $\langle x \rangle : a \longrightarrow a_{\langle y \rangle}$;
2. $\langle y \rangle : a, b \longrightarrow b_{\langle x \rangle}$;
3. $\langle y, !b \rangle : a \longrightarrow a_{\langle x \rangle}$.

These must be applied over three distinct steps and require unique compartment identifiers in order to achieve the outcome of a single antiport rule. EPS communication, on the other hand, is advantageous in scenarios where the minimum between a particular multiplicity and n others is required. This can be computed in parallel having n compartments identified by a y object, each encoding the result as the multiplicity of an object c: $\langle x \rangle : a \longrightarrow a_{\langle y \rangle}$, $\langle y \rangle : a, b \longrightarrow c$. On this basis, n *min* procedures can be computed in two computational steps (irrespective of n).

5.2.4 Permitting and forbidding contexts

Many P system models feature an ancillary control logic, based on elements that are immutable during the execution of a computational step. A promoter (also referred to as activator) is a multiset over a P system's alphabet, ascribed to a particular rule. The presence of the multiset denoted by the promoter in a compartment σ_i enables the execution of this rule and conversely, its absence prohibits the rule from being applied in σ_i. The predicate ensued from this interpretation is evaluated once for each computational step and its boolean outcome persists regardless of how the σ_i's encapsulated multiset is manipulated during one step. A promoter directly translates to an elementary P system scope: $a \longrightarrow b|_z$ is equivalent to $\langle z \rangle : a \longrightarrow b$. Both activator and scope multisets are mutable entities and can be subject to rewriting/communication rules. An inhibitor, however, invalidates the execution of a rule when the multiset it denotes is included in the compartment the rule is associated with: $a \longrightarrow b|_{\neg z}$ is inapplicable if σ_i contains multiset z. Importantly, this does not equate to the EPS scope construction $\langle !z \rangle$ which evaluates each object and its multiplicity in z against a compartment multiset separately (the semantics have been formally defined in the previous chapter). Formally, $z \not\subseteq t \not\to t \equiv \langle !z \rangle$ in elementary P systems. In order to reproduce the behaviour of an inhibitor $\neg z$ on a rule $a \longrightarrow b$, for each object-multiplicity pair $!x_i^{m_i}$ in z we require a rule $\langle !x_i^{m_i} \rangle : a \longrightarrow b$. Alternatively, one may consider a two step procedure whereby the presence of the multiset z in the compartment is first determined: $\langle z \rangle : c \longrightarrow c, t$; subsequently, the rule $\langle !t \rangle : a \longrightarrow b$ can be applied if the multiset z was not encountered. Additionally, the auxiliary object t must be disposed of, if produced: $t \longrightarrow nil$.

5.2.5 Membrane dissolution

Elementary P systems do not feature dissolution rules. We recall that a traditional P system membrane is considered 'dissolved' when a rule with the symbol δ is applied, resulting in the transfer of the compartment's multiset into its parent. By contrast, an EPS compartment is said to be 'detached' from the model when it is no longer addressable by any scope, that is, its associated multiset is empty. In order to achieve this outcome, a communication rule $a \longrightarrow a'_{\langle s \rangle}$ is required for each object a which may identify the compartment. Evidently, the scope s must find at least one congruent compartment which is to receive the content, otherwise a cannot be expelled. Whilst the strategy is highly general, it may be sufficient in some scenarios to employ a reduced number of rules if the multiplicities of objects are constant[1], by grouping objects in the left hand side multiset of a rule. Since EPS compartment detachment is not a distinctive operation with specific semantics, but rather a consequence of a particular local state (a compartment being identified by an empty multiset), the feature has limited functional relevance and thus isolated use cases. Membrane dissolution is essential to most variants having rules statically bound to compartments, immediately pruning execution of a fixed set of instructions. By contrast, elementary P system compartments can be regarded as ‘polymorphic’ - the subset of rules applicable on compartments is re-evaluated each step. This promotes the re-use of operational compartments as opposed to dissolving and creating of new membranes of a different kind (i.e. with a different label).

5.2.6 Membrane division

Advanced by P systems with active membranes (and their extensions), membrane division is one of the most prominent features of the membrane computing paradigm. The parallel execution of membrane division rules can generate an exponential computational space in linear time - a potential extensively investigated for solving NP-complete problems, as we have seen in chapter 3. In addition to expanding the set of compartments of a P system model, division rules will also replicate a compartment’s multiset content (after multiset rewriting and communication rules have been applied) to the emerging membranes. Whilst the number of compartments one can divide into is not restricted to two, it must, however, be constant (i.e. one cannot divide into an arbitrary number of compartments).

Since elementary P systems aspire to a reduced set of primitives with simplified semantics, compartment expansion is effected by rules with structural and semantic similarity to multiset rewriting and communication. Each compartment in the model’s set C is associated a non-operational ‘quiescent’ compartment, which is only instantiated when a multiset is found to identify it. An EPS instantiation rule resembles a typical communication segment (a_{scope}, a_{all}) with a reserved target indicator (denoted by *new* in our formal definition) and is an equal contender to the maximally parallel execution strategy. A single application of such a rule is sufficient to activate the quiescent compartment. There is a single quiescent compartment associated with each operational compartment in C (each computational step) and consequently, EPS compartment expansion rate

[1]This is expected to occur frequently in elementary P system models, since objects have an amplified significance and may denote compartment identifiers, indices, state identifiers, boolean flags etc.

is at most a product of two per step. In contrast to membrane division semantics, the remaining content of a membrane is not replicated to the emerging compartment, nor is the compartment which triggered the instantiation dissolved in the process. In order to effectively copy the content of a compartment σ to another, one may include communication rules of the form $x \longrightarrow x, x_s$, where s is a scope which uniquely identifies the target compartment, and x is an object which may be included in σ. If the emerging compartment is the intended recipient, then we substitute the scope s with the '*new*' indicator: $x \longrightarrow x, x_{new}$. Since rules are dynamically assigned to compartments on the basis of scope congruence, there is no requirement to ascribe a type identifier (i.e. membrane label) or polarisation symbol (+, $-$, 0) to indicate the behaviour of the instantiated compartment. In chapter 8 we demonstrate a linear time solution to the Subset Sum problem using EPS compartment instantiation. The algorithm employs the aforementioned technique for replicating the existing content of a compartment, in addition to emitting distinctive objects (which designate the system's state) to a quiescent compartment.

5.3 The semantics of elementary P systems

Elementary P systems are inherently parallel computational models, featuring primitives which advocate not only parallel multiset rewriting and communication, but also parallel expansion via compartment instantiation. A formal description of these operations by principles and methods immanent to this class of models is a contentious matter and, in the author's opinion, unattainable at this stage. Whilst, in computer science, the 'sequential-parallel' dichotomy may be regarded as a standard characterisation of dynamics in a qualitative sense, the concept of 'parallelism' (in this context) does not bear the same specificity nor rigorous (formal) account of its significance as its counterpart. Rather, 'parallelism' frequently appears as a label to indicate *non-sequential* behaviour, or dynamics of multiple entities in a distributed system. It is through the perspective of sequential computation that parallel transformation is generally perceived and formally defined. Our approach for describing the semantics of elementary P systems is conventional in this respect.

In this section we define the the maximally parallel transition of elementary P systems using structural operational semantics [73], a formal framework first proposed by Gordon Plotkin in the 1981 [72]. This methodology is particularly effective in describing the behaviour of a system in terms of its components. Specifically, individual transitions are defined as a set of inference rules of the form $\frac{premises}{conclusion}$, which compositionally describe the computations of a program as a deductive tree. Structural operational semantics have been successfully used in the context of membrane computing, most notably in [15, 17, 23]. The formal descriptions presented in these studies are all pertinent to traditional P systems, with an extension to accommodate rule priorities documented in [17]. To the best of our knowledge, the (various) strategies advanced in the aforementioned papers have not been applied on any of the P system variants surveyed in this thesis.

Complementary to the operational approach to EPS semantics, the supporting procedures captured by routines such as *rar* (random applicable rule), *apply* (apply a particular rule) or predicates like *isRuleApplicable* are formally described in a denotational style [83]. Both descriptions relate to the same EPS data structure definitions, however, the ancillary procedures are more concisely (and perhaps intuitively) represented using a notation that resembles that of functional programming languages.

We begin with a definition of the abstract syntax of the model's constituents as BNF (Backus-Naur Form)-like statements. We note that many of the structures declared to this end also comprise auxiliary elements which constitute intermediate, local states derived from subsidiary sequential transitions[1].

A multiset can be inductively defined as:

$$\omega ::= \epsilon \mid \omega^+$$
$$\omega^+ ::= a \mid a : \omega$$

where a is a symbol from a finite EPS alphabet which excludes ϵ and ':' denotes the well known *cons* operator.

A compartment is provisioned as a multiset triple:

$$c ::= \langle \omega, \omega_i, \omega_n \rangle$$

where $\omega_i ::= \omega$ represents an *intermediate store* (or multiset buffer) which collects all productions destined for this compartment; and $\omega_n ::= \omega$ is also a multiset accumulator to be *committed* to an new (instantiated) compartment if non-empty.

Next, a list of compartments is described as:

$$C ::= \epsilon \mid c : C$$

A scope is essentially a pair of multisets which are evaluated in predicate functions to establish scope congruence. A targeted multiset is a multiset with an associated scope structure. Targeted multisets represent dedicated segments of

[1]An alternative to distributing auxiliary elements across an EPS structure would be to contrive an isolated *state store* component which accompanies the pristine EPS model. This however, requires its own traversal procedures which would compromise clarity in other ways.

an elementary P system rule featuring multiset communication and are clustered into a list (T).

$scope ::= \langle \omega, !\omega \rangle$

$\omega^{tar} ::= \langle \omega, scope \rangle$

$T ::= \epsilon \mid \omega^{tar} : T$

The syntax of an EPS rule (r) resembles the expression used in our formal definition (section 5.1), however, the multiset productions (i.e. rule right hand side) are condensed into a structure which consists of a multiset targeting the compartment the rule executes on (ω), a set of targeted multisets (T), $\omega^{all} ::= \omega$ to be replicated to all compartments in the model and $\omega^{new} ::= \omega$ a multiset destined for the quiescent compartment. Since EPS rules are not pre-allocated to compartments, we can also group these into a single list (R) and include this in the $e\Pi$ composite.

$r ::= scope : \omega^{+} \longrightarrow \langle \omega, T, \omega^{all}, \omega^{new} \rangle$

$R ::= \epsilon \mid r\ R$

An elementary P system structure encompasses two sets of compartments, C and C_i. This partitioning strategy is instrumental in expressing the *exhaustive execution* of EPS rules: a compartment c is included in C for as long as there are applicable rules on c; if no further rules can be applied, then c will be 'moved' to C_i. In other words, the two sets C and C_i are required to distinguish between 'exhausted' and operational compartments. This distinction is apparent in the first two inference rules (1 and 2) of our semantics. In addition to the set of rules R which is the sole immutable element of eΠ, $\omega_a ::= \omega$ is a multiset which collects all replicated productions of rules with $\omega^{all} \neq \epsilon$. This outcome is globally accumulated and is required for newly instantiated compartments.

$e\Pi ::= \langle C, C_i, R, \omega_a \rangle$

The EPS maximal parallel transition (denoted by $\overset{mp}{\Longrightarrow}$) is defined on the basis of two distinct computational phases:

α : All applicable rules are executed on operational compartments in $e\Pi$'s set C and their productions *buffered* (temporary stored) in the ancillary multisets ω_i, ω_n (for each compartment) and ω_a. This operation continues until all compartments in C have been 'exhausted' (and thus relocated to C') - that is, until C becomes empty (ϵ);

β : All accumulated productions are committed to the compartment's base multiset ω and the supporting multisets (ω_i, ω_n, ω_a) are depleted (i.e. reset). For each compartment with a non-empty ω_n, an additional membrane identified by $\omega_n + \omega_a$[1] is instantiated and added to the set C (of $e\Pi$). Conversely, if a committed compartment remains empty ($\omega = \epsilon$), then it is not further included in C.

1. $$\frac{\langle c : C, C_i, R, \omega_a\rangle,\ rar(R, c, C, C_i) \rightarrow r,\ apply(r, c, C, C_i, \omega_a) \rightarrow \langle c', C', C_i', \omega_a'\rangle}{\langle c : C, C_i, R, \omega_a\rangle \overset{\alpha}{\Longrightarrow} \langle c' : C', C_i', R, \omega_a'\rangle}$$

2. $$\frac{\langle c : C, C_i, R\rangle,\ rar(R, c, C, C_i) \rightarrow \epsilon}{\langle c : C, C_i, R, \omega_a\rangle \overset{\alpha}{\Longrightarrow} \langle C, c : C_i, R, \omega_a\rangle}$$

3. $$\frac{\langle C, C_i, R, \omega_a\rangle \overset{\alpha}{\Longrightarrow} \langle \epsilon, C_i', R, \omega_a'\rangle}{e\Pi \overset{\alpha}{\Longrightarrow} e\Pi_i}$$

4. $$\frac{\langle \epsilon, C_i, R, \omega_a\rangle,\ commit(C_i, \omega_a) \rightarrow C'}{\langle C, C_i, R, \omega_a\rangle \overset{\beta}{\Longrightarrow} \langle C', \epsilon, R, \epsilon\rangle}$$

5. $$\frac{\langle C, C_i, R, \omega_a\rangle \overset{\beta}{\Longrightarrow} \langle C', \epsilon, R, \epsilon\rangle}{e\Pi_i \overset{\beta}{\Longrightarrow} e\Pi'}$$

6. $$\frac{e\Pi \overset{\alpha}{\Longrightarrow} e\Pi_i,\ e\Pi_i \overset{\beta}{\Longrightarrow} e\Pi'}{e\Pi \overset{mp}{\Longrightarrow} e\Pi'}$$

The first three statements (1, 2 and 3) in the above listing describe the α transition of an elementary P system $e\Pi$ to an *intermediate configuration* $e\Pi_i$. This is achieved by an exhaustive execution of all applicable rules in R: if there exists an applicable rule r, randomly selected from R, then this rule will be applied and the compartment is retained in C (as operational). The *rar* (random applicable rule)

[1]The operator '+' signifies multiset addition or concatenation in this expression.

and *apply* functions are formally defined in subsequent listings, in this section. We note that the outcome of *apply* is observed on the compartment the rule is effected on and possibly on other compartments in C or C_i via communication rules, as well as ω_a if the rule channels objects to all existing compartments in $e\Pi$.

The second rule (number 2) infers that a compartment is to be considered 'exhausted' (non-operational) when no further random applicable rule can be found (i.e. *rar* yields ϵ). In this case, the compartment is relocated to C_i. When all compartments in C have been 'exhausted' ($C = \epsilon$) and no other rule in the model can execute, then $e\Pi$ is considered to be migrated to the intermediate configuration (inference number 3).

The following two rules (4 and 5) illustrate the β transition - the *commit* phase during which all multiset productions accumulated are applied to the compartments of $e\Pi$. Furthermore, new compartments ($\omega_n \neq \epsilon$) are also appended to C whilst empty compartments ($\omega = \epsilon$) are pruned from the set. The 'commit' procedure is also described in a denotational style in the following listing.

Finally in rule 6, an EPS maximally parallel computational step is defined as a sequence of the two transitions α and β.

We next present the semantics of the three functions *rar*, *apply* and *commit* using a simplified notation that resembles functional programming languages. There are no explicit type annotations for the parameters or return values of these procedures, however, by convention we use variable names or terms which reflect the syntax definitions presented earlier, where appropriate. Essential predicates, such as *isRuleApplicable* or *isCompartmentInScope* are also included as are all supporting rules, regardless of generality (*concat*, *add*, *subtract* etc). We also note that some of the statements are not declared generically and are only pertinent to this context (i.e. the complete semantics of EPS). For example, multiset subtraction (*subtract*) does not ascertain whether the first operand (to be subtracted *from*) includes (or equals) the second, a condition sine qua non for objects whose multiplicities can only be represented as natural numbers. This is, of course, not required because the evaluation (semantic rule *includes*) precedes all references to this rule, permitting this level of specificity.

Whilst the listing is devoid of obtrusive descriptions, comments prefixed with '#' accompany sections which may be less intelligible or require clarification. In addition to a standard **if ... then ... else ...** clause, we also assume **true** and **false** as the two boolean values, whilst && and || denote the logical **and** and **or** operators, respectively.

```
# Random applicable rule, selected from a set $R$ and
# evaluated on compartment $c$, and sets $C$ and $C_i$
```

```
$rar(R, c, C, C_i) = rarProc(R, c, C, C_i, \epsilon)$

$rarProc(\epsilon, c, C, C_i, r_s) = r_s$
$rarProc(r : R, c, C, C_i, r_s) =$
    $if\ isRuleApplicable(r, c, C, C_i)$
        $then\ rarProc(R, c, C, C_i, ndsA(r, r_s))$
        $else\ rarProc(R, c, C, C_i, r_s)$

# Non deterministic selection when y is not $\epsilon$
$ndsA(x, y) =$
    $if\ y == \epsilon$
        $then\ x$
        $else\ nds(x, y)$

# Non deterministic selection
$nds(x, y) = x$
$nds(x, y) = y$

$isRuleApplicable(scope :\ \omega \longrightarrow \langle \omega, T, \omega^{all}, \omega^{new} \rangle, c, C, C_i) =$
    $isCompartmentInScope(c, scope)$ &&
    $includesMultiset(c, \omega)$ &&
    $atLeastOneRecipientForEachTar(T, concat(C, C_i))$

$isCompartmentInScope(\langle \omega, \omega_i, \omega_n \rangle, scope) = isScopeCongruent(\omega, scope)$

# An empty multiset is (by design) not congruent with any scope
$isScopeCongruent(\epsilon, scope) = false$
$isScopeCongruent(\omega^+, \langle \omega_p, !\omega_q \rangle) =$
    $includes(\omega^+, \omega_p)$ && $doesNotIncludeSegmentOf(\omega^+, \omega_q)$

# Multiset inclusion predicate
$includes(\omega, \epsilon) = true$
$includes(\omega, a : \omega_s) = includesSymbol(\omega, a)$ && $includes(subtractSymbol(\omega, a), \omega_s)$

$includesSymbol(\epsilon, a) = false$
$includesSymbol(a : \omega, a) = true$
$includesSymbol(b : \omega, a) = includesSymbol(\omega, a)$

$subtractSymbol(\epsilon, a) = \epsilon$
$subtractSymbol(a : \omega, a) = \omega$
$subtractSymbol(b : \omega, a) = b : subtractSymbol(\omega, a)$
```

```
# Second condition which must be satisfied for scope congruence
$doesNotIncludeSegmentOf(\omega, \epsilon) = false$
$doesNotIncludeSegmentOf(\omega, a : \omega_q) =$
    $if\ includesSymbol(\omega_q, a)$
        $then\ doesNotIncludeSegmentOf(subtractSymbol(\omega, a), \omega_q)$
        $else$
            $not(includesSymbol(\omega, a))$ &&
            $doesNotIncludeSegmentOf(subtractSymbol(\omega, a), \omega_q)$

$not(true) = false$
$not(false) = true$

$includesMultiset(\langle \omega, \omega_i, \omega_n \rangle, \omega_s) = includes(\omega, \omega_s)$

$atLeastOneRecipientForEachTar(\epsilon, C) = true$
$atLeastOneRecipientForEachTar(t : T, C) =$
    $atLeastOneRecipient(t, C)$ && $atLeastOneRecipientForEachTar(T, C)$

$atLeastOneRecipient(t, \epsilon) = false$
$atLeastOneRecipient(\langle \omega, tar \rangle, c : C) =$
    $isCompartmentInScope(c, tar)\ ||\ atLeastOneRecipient(\langle \omega, tar \rangle, C)$

$concat(C_1, \epsilon) = C_1$
$concat(C_1, c : C_2) = c : concat(C_1, C_2)$

$apply(scope:\ \omega^{lhs} \longrightarrow \langle \omega^{self}, T, \omega^{all}, \omega^{new} \rangle, c, C, C_i, \omega_a) =$
    $let\ c' = applyInstantiation(\omega^{new}, applyRewrite(\omega^{lhs} \longrightarrow \omega^{self}, c))$
    $let\ \langle C', C_i' \rangle = applyAllCommunication(applyTarCommunication(T, C, C_i), \omega_a)$
    $return\ \langle c', C', C_i', add(\omega^{all}, \omega_a) \rangle$

$applyInstantiation(\omega^{new}, \langle \omega, \omega_i, \omega_n \rangle) = \langle \omega, \omega_i, add(\omega^{new}, \omega_n) \rangle$

# Multiset addition, equivalent to string concatenation for
# this representation of a multiset
$add(\omega_1, \epsilon) = \omega_1$
$add(\omega_1, a : \omega_2) = a : add(\omega_1, \omega_2)$

$applyRewrite(\omega^{lhs} \longrightarrow \omega^{rhs}, \langle \omega, \omega_i, \omega_n \rangle) = \langle subtract(\omega^{lhs}, \omega), add(\omega^{self}, \omega_i), \omega_n \rangle$

# Multiset subtraction
```

```
$subtract(\omega_1, \epsilon) = \omega_1$
$subtract(\omega_1, a : \omega_2) = subtract(subtractSymbol(\omega_1, a), \omega_2)$

# Apply communication rules which replicate to all EPS compartments
$applyAllCommunication(C, C_i, \omega) = \langle addToAll(C, \omega), addToAll(C_i, \omega)\rangle$

$addToAll(\epsilon, \omega) = \epsilon$
$addToAll(c : C, \omega) = addToCompartment(\omega, c) : addToAll(C, \omega)$

$addToCompartment(\omega^p, \langle\omega, \omega_i, \omega_n\rangle) = \langle\omega, add(\omega^p, \omega_i), \omega_n\rangle$

$applyTarCommunication(\epsilon, C, C_i) = \langle C, C_i\rangle$
$applyTarCommunication(T, C, C_i) = \langle applyTarList(T, C), applyTarList(T, C_i)\rangle$

$applyTarList(\epsilon, C) = C$
$applyTarList(t : T, C) = applyTarList(T, applySingleTar(t, C))$

$applySingleTar(t, \epsilon) = \epsilon$
$applySingleTar(\langle\omega, scope\rangle, c : C) =$
    $if\ isCompartmentInScope(c, scope)$
        $then\ addToCompartment(\omega, c) : applySingleTar(\langle\omega, scope\rangle, C)$
        $else\ c : applySingleTar(\langle\omega, scope\rangle, C)$

# The rule assumes that $\epsilon : C = C$ such that if there is no
# instantiation and $c = \epsilon$ is returned, then $c : C = C$
$commit(\epsilon, \omega_a) = \epsilon$
$commit(c : C_i, \omega_a) =$
    $commitInstantiation(c, \omega_a)$ :
    $(pruneCompartment(commitCompartment(c)) : commit(C_i, \omega_a))$

$commitInstantiation(\langle\omega, \omega_i, \epsilon\rangle, \omega_a) = \epsilon$
$commitInstantiation(\langle\omega, \omega_i, \omega_n\rangle, \omega_a) = \langle add(\omega_n, \omega_a), \epsilon, \epsilon\rangle$

$pruneCompartment(\langle\epsilon, \omega_i, \omega_n\rangle) = \epsilon$
$pruneCompartment(\langle\omega^+, \omega_i, \omega_n\rangle) = \langle\omega^+, \omega_i, \omega_n\rangle$

$commitCompartment(\langle\omega, \omega_i, \omega_n\rangle) = \langle add(\omega, \omega_i), \epsilon, \epsilon\rangle$
```

Having defined elementary P systems, their features described and exemplified and having established the relevance of this model in the context of membrane computing - morphologically and functionally, we next demonstrate, in chapter 6, the formal verification approach which substantiates this thesis. The polyvalent case studies presented in the following chapters also illustrate modelling principles pertinent to EPS and highlight conversion strategies for other P system variants.

Chapter 6

Model checking Elementary P systems

The formal verification approach set forth in this thesis does not diverge from conventional methodology. One key contribution for achieving this goal is the provision of an abstract formal framework which incorporates the substrate of membrane computing but also translates to an optimal *process model* required by a model checker tool. Complementary, describing P system semantics, particularly its emblematic maximal parallel execution strategy, in terms of concurrent sequential processes according to the principles defined in chapter 4 establishes the feasibility of model checking membrane systems.

The four distinct stages which comprise our approach can be iterated as follows:

1. Formal modelling with elementary P systems;
2. Automatic conversion of the EPS model to a process meta language (Promela) specification required by SPIN;
3. Construction of a set of properties we wish to verify, specified as LTL formulae;
4. Formal verification of the produced formulae against the converted EPS model, using the SPIN model checker.

6.1 Software tools and the eps specification

The methodology proposed and demonstrated in this thesis is accompanied by a suite of tools which mediate the transition between the steps identified in the above listing. Specifically, a library (referred to as *eps-tools*) of software modules centred around elementary P systems was developed to facilitate:

1. Model analysis via guided simulations of an EPS model;
2. Automatic model translation to SPIN's Promela specification according to the principles advanced in this thesis.

In addition, *eps-tools* also integrate a parser for a simple specification language designed to express elementary P systems in a unambiguous and intuitive manner. Both the simulation and conversion programs accept (and require) the eps[1] modelling language as input format.

The restricted set of primitives promoted by elementary P systems is reflected in the concise syntax of the language. An EPS model definition consists of a sequence of statements which are either compartment declarations or rule expressions. Each compartment is composed of a non-empty multiset encoded as a parentheses enclosed, comma separated sequence of object identifiers. These can be prefixed by a numerical value which denotes the object's multiplicity: `(a, 2b, 3c); (x, y); (10 t);`. The absence of a multiplicity indicator implies the presence of a single object.

Each *eps* statement ends with a semicolon ';' whilst spaces within declarations, wrapping operators or multiset values, are ignored by the parser.

A rule is expressed as `scope: lhs -> rhs;`, where *scope* and *lhs* are multiset terms and expanded accordingly. The *rhs* operand is a comma delimited succession of multiset products of the form:

1. `[ms]` corresponds to communication by broadcasting to *all* compartments of $e\Pi$; *ms* is the multiset of objects that is propagated across C.
2. `[ms @ scope]` where *ms* and *scope* are multisets of object; *ms* represents the outcome of the rule whereas *scope* indicates (by congruence) the compartments in C who will receive a copy of *ms*;
3. `[ms *]` denotes a transition of *ms* to the quiescent compartment associated with $c \in C$ where the rule is applied in.
4. `ms` stands for simple multiset rewriting; the product of the rule does not leave the compartment - the implicit target indicator of *ms* is *self*.

We illustrate EPS rule expressions with the following examples, tabulated in 6.1.

[1]We note the symbolic distinction between the lower cased 'eps' and the upper-case 'EPS'. The *eps* term denotes the name of the specification language for elementary P systems and also prefixes the supporting software tools (*eps-tools*, eps simulator, *eps2spin* etc), whereas 'EPS' is a shorthand expression for 'elementary P systems' and is used interchangeably.

EPS rule	Equivalent expression in the eps specification
$a \longrightarrow b, c^3, m_{\langle n \rangle}$	`a -> b, 3c, [m @ n];`
$\langle m, n^2 \rangle : x \longrightarrow y_{all}$	`m, 2n:  x -> [y];`
$\langle m, !b, c^2 \rangle : x, y \longrightarrow x, y_{all}, y_{\langle m, !f \rangle}$	`m, !(b, 2c):  x, y -> x, [y], [y @ m, !f];`
$\langle !z \rangle : y \longrightarrow y_{all}, x_{new}$	`!z:  y -> [y], [x *];`

Table 6.1: EPS rules and their equivalent expression in the eps specification

In addition to prefixing individual rules with scope multisets, the eps language also allows a collection of rules under the same scope to be expressed more succinctly. The construct is referred to as *scope closure* and is nothing more than a syntactic reduction, there is no augmentative logic applied to the conditional element. For instance:

```
e, f: {
  !2f: p -> q;
  m, !z: {
    !n: p -> [q];
    n, !2n: p -> [q *];
  }
}
```

is functionally equivalent to:

```
e, f, !2f: p -> q;
e, f, m, !(n, z): p -> [q];
e, f, m, n, !(z, 2n): p -> [q *];
```

Another significant aspect of this notation is the *inferred alphabet of objects.* A specific declaration of the objects found in compartment multisets, scopes or rule terms is not necessary - the system alphabet is compiled ‘on the fly’.

We conclude our discussion of the eps modelling language with a complete Extended Backus-Naur Form (EBNF) specification in the following listing:

```
eps = {eps statement};
eps statement = [space_c], statement unit
    | statement unit, [space_c], eps statement;

space_c = [space | (comment, space_c)];

space = ‘ ’ | ?line end? | {space};

comment = ‘/*’, anything except comment end, ‘*/’;

anything except comment end = [-‘*/’]
    | (-‘*/’), anything except comment end;

statement unit = compartment | rule | scoped rule | scoped rule set;

compartment = ‘(’, multiset, ‘)’;

multiset = multiset unit
    | multiset unit, [space_c], ‘,’, [space_c],  multiset;

multiset unit = [number], identifier;

number = digit - ‘0’
    | digit - ‘0’, {digit};

digit = ?0-9?;

identifier = letter
    | identifier, [letter, digit | ‘_’];

letter = ?a-zA-Z?;

rule = multiset, [space_c], ’->’, [space_c], (multiset | target multiset);

target multiset = ‘[’, [space_c], multiset, [space_c], [target], ‘]’;

target = ’@’, [space_c], (scope | ‘*’);

scope = scope unit
    | scope unit, [space_c], ‘,’, [space_c], scope;

scope unit = multiset unit
```

```
    | ‘!’, [space_c], multiset unit
    | ‘!’, [space_c],  ‘(’, multiset, ‘)’;

scoped rule = scope, [space_c], ‘:’, rule;

scoped rule set = scope, [space_c], ‘:’, [space_c], ‘{’,
    {[space_c], rule statement, [space_c]}, ‘}’;

rule statement = (rule | scoped rule), [space_c], ‘;’
    | scoped rule set;
```

Whilst an EPS model simulator is of great convenience when designing more elaborate algorithms and generally for analysing the behaviour of a modelled system, the key utility featured by our library is the EPS translation tool, which automatically generates a process model (Promela specification) suitable for SPIN model checking. The software program implements a relatively simple model transformation, which is indeed a merit of the minimal set of primitives employed by elementary P systems. The so called *state vector* comprises a single array of *compartment* data structures. These envelop two lists of length `A_SIZE`, that is, the size of the alphabet of objects O. Each index in a list represents a multiset object whereas the value at index i equates to the multiplicity of the object encoded by i in a particular compartment. The Promela code below illustrates this structure and also includes the declaration of the compartment array.

```
typedef Compartment {
     short x[A_SIZE] = 0;
     short y[A_SIZE] = 0;
     bit rulesApplicable[RULE_COUNT] = 0;
}
Compartment C[MAX_COMPARTMENTS];
short C_COUNT = 3;
```

An additional auxiliary element in the type definition above is a list of boolean variables whose values indicate which rules are applicable to a particular compartment at a certain step. These are computed and stored before the rules are selected for application, at the beginning of each step. The `RULE_COUNT` and `MAX_COMPARTMENTS` constants symbolise the number of rules the P system employs and the maximum number of compartments defined for this experiment, respectively. For models with no compartment instantiation rules, `MAX_COMPARTMENTS` will be equal to `C_COUNT`, which represents the initial number of compartments in the array `C`.

Scopes and rules are directly translated into C code and interspersed with the Promela declarations and the core functional unit. This is a fruitful feature offered by the SPIN model checker: not only is the C code seen as atomic instruction blocks by SPIN, but it also confers flexibility for implementing more complex procedures in a familiar programming language.

Each scope encountered in the EPS model is a predicate effectively referenced by index in the C code generated. For example, the following is a standard C function which returns *true* if the compartment identified by `cIndex` contains at least one b object and *false* otherwise.

```
bool isInScope0(short cIndex) {
    short *x = now.C[cIndex].x;
    return (x[_b] >= 1);
}
```

Such a routine ($isInScope_i$) is created for every scope in $e\Pi$, regardless of whether it is used for rule allocation or as a target indicator. Compound predicates can be subsequently queried at a higher level. For instance:

```
bool isRule1ApplicableTo(short cIndex) {
    return isInScope1(cIndex)
        && atLeastOneInScope2(cIndex);
}
```

asserts whether a rule with associated index 1 can be applied in a compartment whose index is passed by the parameter `cIndex`. Since communication rules require at least one valid (scope congruent) recipient, a second predicate is necessary to determine the applicability of this rule. This is reflected by the latter term of the conjunction, `atLeastOneInScope2(cIndex)`[1].

Scope predicates are also referenced in rule application procedures. The following listing demonstrates a communication rule which extracts one a object from the `cIndex` compartment and sends a copy of a to all membranes congruent with some scope whose index is 2.

```
void applyRule1(short cIndex) {
    now.C[cIndex].x[_a] -= 1;
    for(short i = 0; i < now.C_COUNT; ++i) {
```

[1]The argument `cIndex` is required to exclude the compartment identified by this index from the scope congruence search. At least one region *other then cIndex* congruent with scope `Scope2` must exist to validate the applicability of `Rule1`.

```
        if(i != cIndex) {
            if(isInScope2(i)) {
                now.C[i].y[_a] += 1;
            }
        }
    }
}
```

We note the `now` identifier which is simply a reference to the state vector, provided by SPIN. We recall our `Compartment` data type definition which includes two arrays, `x` and `y`. Whilst `x` represents the multiset currently present in compartment `cIndex`, `y` acts as a buffer which temporarily stores the multiset productions destined to this compartment. More precisely, all rewriting and communication rules with a yield for `cIndex` will provisionally store this yield in `y` and after all possible rules have been applied during the respective step, the vector `y` is committed to the persistent multiset encoded by `x`. `y` is immediately reset (filled with 0) after each commit. This strategy is strictly necessary in the implementation of maximal parallelism as a sequential process.

In order to make the generated code more legible, the objects of the alphabet O are denoted by more intuitive identifiers, specified as macros. These are always successive numbers starting from 0, as they are utilised as indices for the vectors `x` and `y`.

```
#define _a 0
#define _b 1
#define _c 2
```

To reduce the risk of a naming collision, the convention adopted for object identifiers is the underscore prefix.

The core of the EPS execution algorithm tailored for the SPIN model checker is best illustrated by a pseudo-code representation. This is depicted in listing 6.1 and concludes our description of the EPS to Promela translation strategy. In this listing, we draw attention on the pre-computed set of applicable rules R_i for each compartment i of an EPS (line 4). On the one hand, this is an important optimisation which prevents the unnecessary repetitive evaluation of the 'applicable' predicate on the global set of rules R - rule applicability is subsequently determined in the scope of R_i. On the other hand, the pre-computed set R_i is also utilised to ascertain whether a rule contends (*competes*) with another for execution. Two or more rules are considered to compete with each other for execution in a compartment if and only if:

1. the rules are applicable in the respective compartment and

2. the intersection of their left hand side multisets is non-empty.

For example, rules $a \longrightarrow b$ and $a, b \longrightarrow c$ are contenders for object a in a multiset (a, b, c), whereas $c \longrightarrow a, b, k$ does not compete with the first two for execution. If a rule can be executed independently of other rules from the same set R_i, then this rule will be applied exhaustively (i.e. as long as it remains applicable). This consideration further prunes away the states generated as a result of inconsequential interleaving[1] of atomic instructions: non-deterministic choice is only required for competing P system rules.

We also note that a complete code listing as generated by our software tool is provided in the appendix of this thesis.

```
for each computational step {
  execute the following as an atomic instruction {
    for each compartment c_i in C {
      establish the set of applicable rules R_i for c_i;
      repeat {
       non-deterministically select an applicable rule from R_i:
          if such a rule exists {
            if the rule does not compete with any other rule from R_i
              then apply rule exhaustively;
            else
             apply rule once;
          } else
            break repeat and continue with the next compartment in C;
      }
    }

    commit buffered productions to respective compartments in C;
  }
}
```

Figure 6.1: Pseudo-code summarising the behaviour of the algorithm generated for SPIN model checking.

The *eps-tools* library has been implemented using Javascript [7] in its entirety and is based on the Node.js [3] runtime which, importantly, ensures consistent cross-platform operability. The code developed can easily be adapted to run in a browser with an HTML architectured graphical user interface. Not only is

[1]Interleaving is inconsequential when it yields spurious system states. Strictly speaking, the outcome is severely counter-productive, particularly when model checking is a set objective.

Javascript one of the most versatile scripting languages at this time, but also its related JSON [2] notation offers a significant advantage for concise specifications and portability. The initial version of our simulator required an EPS model be expressed as a JSON document, a feature still retained in *eps-tools*, although the preferred input is now the eps modelling language.

6.2 Generating the Fibonacci sequence of numbers

We start with a simple example whose purpose is manifold:

1. Illustrate elementary P systems and
2. The eps modelling language;
3. Demonstrate the *epss* simulator and
4. *eps2spin* translation tool;
5. Introduce the SPIN based model checking method on EPS;
6. Emphasize the difference between standard and efficient modelling of maximally parallel multiset rewriting in SPIN's process meta language;

We consider the following elementary P system:

$$e\Pi_{fib} = (O, C, R)$$

where

$$O = \{a, b, c, x\}$$

$$C = \{(a, x), (b, x), (c)\}$$

$$R = \{\langle b\rangle : x \longrightarrow x, x_{all};\ \langle c\rangle : x \longrightarrow x_{\langle b\rangle}\}$$

An initial remark on this model is the use of objects a, b, c as compartment identifiers. We underline the absence of membrane labels whose role is substituted by the multiset contained in the compartment. a, b and c are constant throughout the execution of $e\Pi_{fib}$ since they are not subject to any rules and hence, never transformed. Rule $\langle b\rangle : x \longrightarrow x, x_{all}$; will always be applied in compartments containing at least one b and in this example (b, x) satisfies this condition *each computational step.*

On the same note, we can infer that a is not strictly necessary since there are no rules which address compartments with a objects, nor are there any communication rules with scopes relating to a. There is indeed no functional necessity for a however, its inclusion confers clarity on the model, specifically when referring to the compartment it is contained by. For convenience, we will subsequently address the three compartments in C with scoped constructs $\langle a \rangle$, $\langle b \rangle$ and $\langle c \rangle$[1].

The execution of $e\Pi_{fib}$ proceeds as follows:

Step 1 : Rule $\langle b \rangle : x \longrightarrow x, x_{all}$ is applied once in compartment (b, x) since a single x is present. The rule yields one x which is retained inside the membrane and another x is replicated to all other compartments, that is (a, x) and (c). The absence of x objects in $\langle c \rangle$ prohibits the execution of rule $\langle c \rangle : x \longrightarrow x_{\langle b \rangle}$. $M_1^{e\Pi_{fib}} = \{(a, x^2), (b, x), (c, x)\}$;

Step 2 : Rule $\langle b \rangle : x \longrightarrow x, x_{all}$ executes in the same manner as in the preceding step, however rule $\langle c \rangle : x \longrightarrow x_{\langle b \rangle}$ will also be applied once, given the presence of an x in $\langle c \rangle$. The outcome of this rule is an x emitted to all compartments containing at least one b - in this example the recipient is a singleton. $M_2^{e\Pi_{fib}} = \{(a, x^3), (b, x^2), (c, x)\}$;

Step 3 : Rule $\langle b \rangle : x \longrightarrow x, x_{all}$ is applied twice during this step, expending the two xs and broadcasting two copies to $\langle a \rangle$ and $\langle c \rangle$; rule $\langle c \rangle : x \longrightarrow x_{\langle b \rangle}$ executes once, sending an x to $\langle b \rangle$. $M_3^{e\Pi_{fib}} = \{(a, x^5), (b, x^3), (c, x^2)\}$;

Table 6.2 lists the first five configurations of $e\Pi_{fib}$.

An immediate observation is that $e\Pi_{fib}$ cannot have any halting computations. The two rules will be applied indefinitely, generating successive numbers in the Fibonacci series. The following relation can be identified in the sequence of configurations illustrated so far: $|\langle a \rangle|_x = |\langle b \rangle|_x + |\langle c \rangle|_x$. More plainly, the number of x objects in the compartment identified by scope $\langle a \rangle$ is equal to the sum of the xs in $\langle b \rangle$ and $\langle c \rangle$ respectively. As such, the three compartments hold sequences of size three from the Fibonacci series of numbers.

To verify these observations we first express our EPS model using the eps language syntax (Fig. 6.2).

To simulate the specified model, the following command is run from a command line terminal:

```
node epss Fibonacci.eps -n 10
```

[1]Scope congruence is always applied by EPS on its entire set of compartments C and generally equates to a set of compartments. In unequivocal instances where it is always a single compartment that is congruent to a scope σ, then we will refer to this compartment by σ.

Step/Compartment	$\langle a\rangle$	$\langle b\rangle$	$\langle c\rangle$
0	a, x	b, x	c
1	a, x^2	b, x	c, x
2	a, x^3	b, x^2	c, x
3	a, x^5	b, x^3	c, x^2
4	a, x^8	b, x^5	c, x^3
5	a, x^{13}	b, x^8	c, x^5

Table 6.2: Execution trace of an EPS based algorithm for generating the Fibonacci sequence of numbers

```
(a, x);
(b, x);
(c);

b: x -> x, [x];
c: x -> [x @ b];
```

Figure 6.2: $e\Pi_{fib}$ expressed in the eps modelling language

where *Fibonacci.eps* is the file name for the implementation of $e\Pi_{fib}$ and *-n* is a parameter which denotes the number of steps to simulate. Other parameters allow for various output formats (in addition to plain text) and fine grained detail in its reporting (an option known as verbosity). A complete description of the tool and its options is available at [1]. The plain text output generated by the aforementioned command, that is, a ten step computation of $e\Pi_{fib}$ is provided in appendix A.

A fully automated conversion to a Promela specification is achieved by executing:

```
node eps2spin Fibonacci.eps -o Fibonacci.pml
```

where the *-o* parameter followed by *Fibonacci.pml* designates the output file the model is generated to. For reference, we include a complete listing of the generated Promela code for this particular example, in appendix B.

Next, the properties expressed as LTL formulae are appended and the *.pml* (Promela) file supplied as input to the SPIN executable:

```
spin -run -a Fibonacci.pml
```

All experiments presented in this thesis have been performed using SPIN version 6.4.3 on a 2.4 GHz Intel Core i7 MacBook Pro with 8GB of DDR3 RAM.

Property	There exists an $e\Pi_{fib}$ computation which generates the value 21 (the sixth number in the Fibonacci series)				
LTL	F $\|\langle a\rangle\|_x = 21$				
SPIN LTL	`<> (C[0].x[_x] == 21)`				
Description	The state where the multiplicity of object x in the compartment 0 ($\langle a\rangle$) is eventually reached.				
Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.730 MB	8	88 bytes

Table 6.3: Property $P_{fib}1$

The initial two verification instances (Tables 6.3 and 6.4) attest the existence of specific numbers in the set of computable values of $e\Pi_{fib}$. The significance of these experiments is not derived from the properties submitted, but from the possibility to prove their validity against a system with infinite states. We recall that $e\Pi_{fib}$ never halts, its computation of Fibonacci numbers is perpetual. For this particular model, the non-terminating computation translates to an infinite number of states, which appears as an infringement of a basic requirement of the model checking technique. SPIN, however, generates its state graph 'on the fly' which implies that only states associated with a relevant computational path will be created. This strategy not only confers great efficiency to the verification process, but also makes the investigation of so called *reachability properties* feasible. Generally, in order to prove the validity of a property (expressed as an NDFA), it is sufficient to traverse a subset of the model's possible states, *if and only if the property is faithful to the model.* The search remains exhaustive, all paths must be visited, however, it need not descend to infinite depth in the state space when the temporal logic formula does indeed hold.

The following four properties (Tables 6.5, 6.7, 6.8 and 6.9) assert the validity of our model more generally. The leading G (globally) operator requires all system states be visited in order to prove the constancy of a relation. $e\Pi_{fib}$,

Property	There exists an $e\Pi_{fib}$ computation which generates the value 34 ($Fib(7)$) and the preceding values generated are less than or equal to 21 ($Fib(6)$).
LTL	$\lvert\langle a\rangle\rvert_x \leq 21$ U $\lvert\langle a\rangle\rvert_x = 34$
SPIN LTL	`(C[0].x[_x] <= 21) U (C[0].x[_x] == 34)`
Description	The state where the multiplicity of object x in the compartment 0 ($\langle a\rangle$) is eventually reached and in all preceding states, this multiplicity cannot be more than 21. The property effectively states that there is no other number generated between 21 and 34, *before the multiplicity 34 is reached*. Importantly, the verification result does not guarantee that values in this range are not computed beyond this step, that is, model checking is impervious to the monotony of a computable function.

	Evaluation	Time	Memory	States	State vector
Result	True	0s	128.730 MB	9	88 bytes

Table 6.4: Property $P_{fib}2$

however, exhibits an infinite number of states and as such we can only verify the aforementioned properties on a finite subset of states from $e\Pi_{fib}$'s infinite set. Since this may be a frequent requirement in specific contexts, the *eps2spin* conversion tool allows the introduction of a *global restriction* which is embedded in the generated Promela code, but not required in the EPS model. The $-n$ parameter effectively limits the computation to a set number of steps:

```
node eps2spin Fibonacci.eps -o Fibonacci.pml -n 20
```

The resulting configuration features a slice of the infinite state space and can thus be verified exhaustively by LTL formulae which entail 'always' or 'never' claims. Evidently, the interval between the computational states can be manually adjusted to reflect a segment of interest; for instance, it may be important to investigate computation between steps 50 and 100.

Properties $P_{fib}3, 4, 5, 6$ have been successfully validated by SPIN for a 20 step computation of $e\Pi_{fib}$. In table 6.6 we list the experimental results obtained in the case of $P_{fib}3$ with an extended 30, 40 and 50 computational steps. It can be observed that the number of states generated by the model checker is equal to $2 \times$ *step_count* $+ 5$, that is linear to the number of P system states. We also note the constancy of the time and memory required to conduct the verification process, confirming the scalability of the approach for this scenario. Rather, it is SPIN's integer data type 32 bit representation which limits the scope of this investigation (32 bit integer is the largest numeric type available in Promela).

Property	It is always the case that the Fibonacci relation holds for any sequence of three numbers generated by $e\Pi_{fib}$ (in the respective compartments).				
LTL	G $\lvert\langle a\rangle\rvert_x = \lvert\langle b\rangle\rvert_x + \lvert\langle c\rangle\rvert_x$				
SPIN LTL	`[] (C[0].x[_x] == C[1].x[_x] + C[2].x[_x])`				
Description	The *globally* operator ensures that for all computational paths, that is, all states generated by SPIN, the aforementioned relation holds. The leading number computed by $e\Pi_{fib}$ is always encoded as the multiplicity of object x in $\langle a\rangle$, its predecessor in $\langle b\rangle$ and the second preceding number in $\langle c\rangle$.				
Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.730 MB	45	88 bytes

Table 6.5: Property $P_{fib}3$

Step count	States	Time	Memory	State vector
30	65	0s	128.730 MB	152 bytes
40	85	0s	128.730 MB	152 bytes
50	105	0s	128.730 MB	152 bytes

Table 6.6: Verification of $P_{fib}3$ for 30, 40 and 50 maximum computational steps.

A noteworthy observation is the constancy of the number of unique states generated by SPIN for $P_{fib}3-5$. This is reported as 45, despite the fact that some properties can never be invalidated past a certain state (for instance, the number 1597 can never again be generated after step 15). The behaviour, however, is consistent with the model checking technique, namely all possible states of the model must be addressed; there are no inferences derived from the construction of the model, such as the strictly increasing monotony of the function which generates each multiplicity, which could prune the superfluous search operation.

Property	$e\Pi_{fib}$ never computes numbers in the 144 ($Fib(10)$) - 233 ($Fib(11)$) interval.
LTL	G $\lvert\langle a\rangle\rvert_x \leq 144 \vee \lvert\langle a\rangle\rvert_x \geq 233$
SPIN LTL	`[] (C[0].x[_x] <= 144 || C[0].x[_x] >= 233)`

Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.730 MB	45	88 bytes

Table 6.7: Property $P_{fib}4$

Property	The 15th number in the Fibonacci series is 1597.
LTL	G (step = 15 $\rightarrow \lvert\langle a\rangle\rvert_x = 1597$)
SPIN LTL	`[] ((step == 15) -> (C[0].x[_x] == 1597))`
Description	The property guarantees that in all cases (on all computational paths), $e\Pi_{fib}$ computes the multiplicity 1597 during step 15.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.730 MB	45	88 bytes

Table 6.8: Property $P_{fib}5$

This is one of the advantages other formal verification methods (i.e. automated theorem proving) feature, in contrast to model checking.

Property $P_{fib}6$ (Table 6.9) is only relevant if the computation exceeds the number steps required for generating Fibonacci numbers greater than the one of interest (2000000). Indeed, the `MAX_STEPS` constant which restricts computation to the number of steps indicated by its value, has been specified as 50 for this property. Furthermore, the `short` datatype to which variables are bound to by default in the *eps2spin* translation has been upgraded to `int`, a necessary substitution required for representing numbers greater than 32767. This modification is reflected in the state vector which has increased to 152 bytes (76 percent).

We next amend our initial $e\Pi_{fib}$ model and include a compartment whose role is to count the number of steps elapsed (analogously to the Fibonacci example presented in chapter 2). This is achieved by accumulation of n objects, one generated per step. Since we wish to isolate the emission of x objects from $\langle b\rangle$, we also change rule $\langle b\rangle : x \longrightarrow x, x_{all}$ to $\langle b\rangle : x \longrightarrow x, x_{\langle !p\rangle}$, that is x will only

Property	2,000,000 is not a Fibonacci number.
LTL	G $\lvert\langle a\rangle\rvert_x \neq 2000000$
SPIN LTL	`[] (C[0].x[_x] != 2000000)`
Description	Given a finite state vector, it is convenient to formally determine whether a value is part of a function's codomain.

	Evaluation	Time	Memory	States	State vector
Result	True	0s	128.730 MB	105	152 bytes

Table 6.9: Property $P_{fib}6$

be replicated to compartments without p objects. The system's alphabet now includes the two additional symbols p, n: $O = \{a, b, c, x, p, n\}$. The updated model, $e\Pi'_{fib}$,is illustrated by the eps code in Fig. 6.3.

```
(a, x);
(b, x);
(c);
(p);

b: x -> x, [x @ !p];
c: x -> [x @ b];
p -> p, n;
```

Figure 6.3: Modified $e\Pi_{fib}$ eps model to include a counter compartment

We highlight the use of catalyst p in the rule $p \longrightarrow p, n$ (which requires no scope since p is always available in one compartment only). This constitutes a design pattern for modelling singular (or a fixed number of) transitions per computational step in elementary P systems. Generally, when a rule is required to execute no more than k times each step, a catalyst object with multiplicity k can be employed to limit this execution to this end. If $k = 1$, then the rule is effectively applied in the *minimal parallelism* (*min*) process mode. If arbitrary execution is required (*par*), then additional rules can be utilised to define non-deterministic mutations of k during each step. For instance, $p \longrightarrow p^2$; $p \longrightarrow p$; $p^2 \longrightarrow p$, when executed in maximal parallel mode can introduce an arbitrary number of p objects within a specific (extensible) range.

The principal purpose of $e\Pi'_{fib}$ is to demonstrate the absence of interleaving from our translated P system model. It is precisely this aspect of our approach which eliminates the exponential state space expansion which arises due to the manner in which parallel instructions are considered by process algebras. Earlier in chapter 4, we have showed this to be inadequate for modelling maximally parallel transitions customary to P systems. The state space could still grow exponentially if the system exhibits non-deterministic behaviour, however, *not* because parallelism is reduced to non-deterministically distributed execution of processes, which translates to interleaved system states.

In this respect, we remark the same number of states generated by SPIN when model checking property F $|\langle a\rangle|_x = 21$ against both $e\Pi_{fib}$ and $e\Pi'_{fib}$ (Table 6.10), despite the increase in compartments and number of rules. Whilst the state vector reflects the memory required by the additional compartment and objects p and n (56 bytes), the number of states (8) is identical to the one reported for the very first experiment ($P_{fib}1$, Table 6.3).

Property	There exists a computation which generates the multiplicity 21 (the sixth number in the Fibonacci series)				
LTL	F $\lvert\langle a\rangle\rvert_x = 21$				
SPIN LTL	`<> (C[0].x[_x] == 21)`				
Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.730 MB	8	144 bytes

Table 6.10: Property $P_{fib}7$

Finally, property 6.11 asserts the reachability of the state when the tenth number in the Fibonacci series is computed and this occurs precisely after step ten and no sooner. We highlight the difference between a model where transitions are interleaved during a single step[1] and our approach by running experiment $P_{fib}8$ again, however, this time rule applications are not interpreted as a single atomic procedure, but rather they are allowed to generate system states. The time required is insubstantial for such a small example, however the number of states generated by SPIN is 644, that is 52 times higher than that of our original model.

[1]The number of states generated if all instructions across all system compartments are interleaved and each rule application generates a system state is much higher. In this example, we emphasize the effect of intra-compartmental interleaving which is restricted to a single step.

Property	The tenth number in the Fibonacci series will eventually be computed and no greater number is generated up to this point.
LTL	$(\|\langle a\rangle\|_x < 144 \wedge \|\langle p\rangle\|_n < 10)$ U $(\|\langle p\rangle\|_n = 10 \wedge \|\langle a\rangle\|_x = 144)$
SPIN LTL	`(C[0].x[_x] < 144 && C[3].x[_n] < 10) U (C[3].x[_n] == 10 && C[0].x[_x] == 144)`
Description	The LTL formula can be translated as: the multiplicity of n and x, in compartments $\langle a\rangle$ and $\langle p\rangle$, will remain lower than 144 and 10, respectively, until the point where the tenth number in the Fibonacci series is computed; this equates to a state where both $\|\langle a\rangle\|_x$ is 144 and $\|\langle p\rangle\|_n$ equals 10.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.730 MB	12	144 bytes

Table 6.11: Property $P_{fib}8$

Chapter 7

Structured modelling with elementary P systems: counting the number of child nodes in a DAG

7.1 Objectives

In order to demonstrate the modelling potential of elementary P systems, but also highlight translation strategies and design patterns pertinent to the representation of other membrane system variants, we consider the Hyperdag P system example illustrated in section 3.5 of chapter 3. We recall the algorithm originally presented in [81] is a parallel child-node counting technique using ‘broadcasting’ rules in a DAG structure. Since EPS communication rules emit multisets of objects by replication to all eligible (by scope congruence) compartments, it can be concluded that EPS rules are a generalisation of broadcasting rules which may only target *child*, *parent* or *sibling* nodes. These directional labels are based on the (permanent) parent-child relation denoted by the edges of the DAG. By contrast, an EPS has no provision for supplementary elements to explicitly designate a graph, tree or DAG like structure of compartments, and as such, a structure can only be inferred by the set of communication rules assigned to a model.

It turns out that elementary P systems are at least as efficient in modelling a DAG-like distribution of nodes (compartments) as Hyperdag P systems, and by implication all variants which feature a graph (or more specific) membrane structure.

In this chapter we present an EPS node counting algorithm which echoes the example of section 3.5. Not only do we emphasize the versatility and efficiency of EPS modelling by preserving the intuitive node - compartment mapping, a restrictive set of rules and dismissing the extraneous set of edges (which Neural-like, Hyperdag, Tissue, Population, Kernel P systems employ), but also demonstrate our formal verification approach on a structured model[1].

7.2 EPS Model

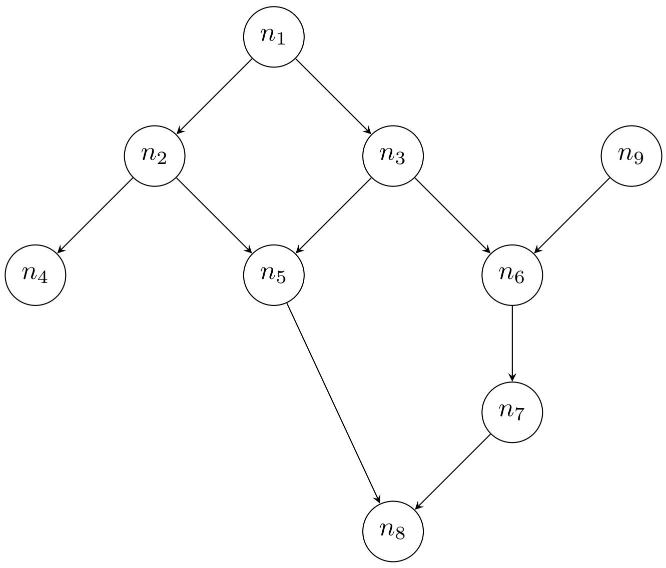

Figure 7.1: A DAG structure modelled by $e\Pi_{dag}$

We consider the following elementary P system for an instance of the problem with $N = 9$ nodes connected as illustrated by Figure 7.1 and $1 \leq i \leq N$:

$$e\Pi_{dag} = (O, C, R)$$

where

$O = \{n_i, a, c, s, q, Pn_i, Cn_i\}$;

$C = \{$
$(n_1, s, q, a, Pn_2, Pn_3)$,
$(n_2, s, q, Pn_4, Pn_5, Cn_1)$,

[1]By structured model, it is referred to a model which faithfully maps a certain arrangement or distribution of entities in an abstract manner. Evidently, all computational models can be considered intrinsically structured (i.e. a sequence of elements can be regarded as a structure due to the regularity of its distribution), however this does not entail *structure* modelling.

$$
\begin{aligned}
&\quad (n_3, s, q, Pn_5, Pn_6, Cn_1),\\
&\quad (n_4, s, q, Cn_2),\\
&\quad (n_5, s, q, Pn_8, Cn_2, Cn_3),\\
&\quad (n_6, s, q, Pn_7, Cn_3, Cn_9),\\
&\quad (n_7, s, q, Pn_8, Cn_6),\\
&\quad (n_8, s, q, Cn_5, Cn_7),\\
&\quad (n_9, s, q, a, Pn_6)\\
&\};
\end{aligned}
$$

$$
\begin{aligned}
&R = \{\\
&\quad \langle n_i \rangle : \{\\
&\qquad q, a \longrightarrow a_{\langle Cn_i \rangle};\\
&\qquad \langle a \rangle : s \longrightarrow c_{\langle Pn_i \rangle};\\
&\quad \}.\\
&\}
\end{aligned}
$$

Similarly to the example in the precedent chapter, objects $n_i, 1 \leq i \leq N$ are included in each compartment and are representative of the node they model. A uniform set of rules R is defined by scope reference to n_i.

The parent-child relations which confer the DAG-like membrane structure to $e\Pi_{dag}$ are denoted by specific objects in O: for each ordered pair (i, j) which represents an arrow in our illustration, we include a Pn_j object in node i and a Cn_i object in node j. This inclusion of Pn_j and Cn_i in the respective compartments marks node i as a parent of j and, conversely, compartment j as a child of node i. In $e\Pi_{dag}$ we can identify node n_1 as the parent of n_2 and n_3, by the presence of objects Pn_2 and Pn_3 in $\langle n_1 \rangle$, but also the inclusion of one Cn_1 object in both $\langle n_2 \rangle$ and $\langle n_3 \rangle$.

In this particular example, a static structure is exercised, however, the mapping facilitates structural mutations by simple transformation (rewriting) of Pn and Cn objects. Moreover, a bi-directional labelling scheme is not strictly required; it may be sufficient in some instances to only use Cn_i ('child of n_i') objects which imply the parent n_i or conversely, Pn_i ('parent of n_i'). In such cases, less data is required to encode the link between two compartments, in contrast to the traditional requirement of a pair (i, j) in a set E of edges. On the other hand, an arbitrary number of connections between two nodes can be conveniently specified, by employing additional objects from O: Ls_k and Le_k may be used to represent the *start* and respectively, the *end* of a link *of type* k between two distinct compartments.

Having defined the initial configuration of $e\Pi_{dag}$ with a consistent mapping scheme, the rule set can be expressed concisely, using the same index $i, 1 \leq i \leq N$ denoting an iteration of length N. Specifically, there are two rules associated

Step/Scope	$\langle n_1 \rangle$	$\langle n_2 \rangle$	$\langle n_3 \rangle$	$\langle n_4 \rangle$	$\langle n_5 \rangle$	$\langle n_6 \rangle$	$\langle n_7 \rangle$	$\langle n_8 \rangle$	$\langle n_9 \rangle$
0	s, q, a	s, q	s, q	s, q	s, q	s, q	s, q	s, q	s, q, a
1	s	s, q, a	s, q, a	s, q	s, q	s, q, a	s, q	s, q	s
2	s, c^2			s, q, a	s, q, a^2		s, q, a	s, q	s, c
3	s, c^2	c^2	c^2	q, a	a	a, c		a^2, s, q	s, c
4	s, c^2	c^2	c^2	q, a	a, c	a, c	c	a^2, q	s, c

Table 7.1: Execution trace of $e\Pi_{dag}$

with each compartment i, one which emits a single a object to direct descendants $q, a \longrightarrow a_{\langle Cn_i \rangle}$ and a second which sends an acknowledgement (object c) to all parent nodes when 'visited' by a: $\langle a \rangle : s \longrightarrow c_{\langle Pn_i \rangle}$. Strictly speaking, the model does not consist of two rules only, but rather of $2 \times N$ rules of the same form. This provision is expanded iteratively by simple substitution of i in n_i, Cn_i, Pn_i with literals in the range [1 .. N]. An elaborated listing of $e\Pi_{dag}$ expressed in the eps modelling language is available in appendix C.

Using objects to encode connections between compartments may not be the most efficient strategy in all circumstances. It is certainly convenient for models with a small number of nodes (such as the DAG in Fig. 7.1) since this permits a more compact rule expression, however if the problem scales to a million nodes or more, then memory considerations are not to be neglected particularly because these ancillary objects will be part of the state vector in the translated Promela model. More precisely, the approach is not scalable although the DAG structure is static. For such large models, minimising the size of the state vector is imperative.

Elementary P systems offer flexibility in this respect, by means of communication rules with scope resolution. A rule $\langle a \rangle : s \longrightarrow c_{\langle Pn_i \rangle}$ can be translated to k rules of the form $\langle a \rangle : s \longrightarrow c_{\langle n_k \rangle}$, where n_k is a parent node of $\langle n_i \rangle$ where the rule executes in. Whilst having a larger set of rules will reduce the performance of the procedure generating the system states, the set remains constant and can be pre-computed for static structures. Rather than referencing relation indicators (Pn_i, Cn_i), the rules target individual nodes directly, invalidating the requirement for additional objects in O and implicitly in the model's state vector. Any decrease in the size of a system state is of consequence and it is not always necessary to resort to a drastic (or optimal) reduction. A hybrid approach may also be adequate and can be achieved for example, by including either a 'parent of' symbol (Pn_i) or 'child of' symbol (Cn_i) in the relevant compartments, as described earlier.

Computation in $e\Pi_{dag}$ proceeds with the application of the first rule, $q, a \longrightarrow a_{\langle Cn_i \rangle}$ in root nodes (i.e. nodes without a parent), since these own an a object in the initial configuration. The same rule executes during each step in each compartment *with at least one* child node, advancing deeper in the hierarchy. Simultaneously, the presence of a also triggers rule $\langle a \rangle : s \longrightarrow c_{\langle Pn_i \rangle}$ in compartments with *at least one* parent node, the effect of which is to emit a c object to all parents of the respective compartment. We underline the use of auxiliary q and s objects with the purpose of *minimising* the execution of the two rules. Indeed, it is always the case that each rule, if applicable, will execute once and once only, regardless of the number of a objects present in the compartment. This approach represents a pattern useful for enforcing *singular execution* of a multiset transition rule.

We remark that 'acknowledgement' rules will never execute in root nodes and complementary, 'broadcasting' rules are inapplicable in leaf (terminal) nodes.

The four step simulation of $e\Pi_{dag}$ is listed in Table 7.1. The system halts after exactly four steps and, more generally, a halting configuration is observed after H steps, where H is the height of the DAG. We note that only the volatile objects have been included in listing 7.1 for a more concise representation; the node markers and parent-child indicators are permanently present during the computation. The outcome of $e\Pi_{dag}$ is denoted by the multiplicity of object c in each node. More precisely, a halting configuration will consist of a number of c objects that is equal to the number of child nodes, in each compartment $\langle n_i \rangle$ in C.

7.3 Verification results

The following properties formally assert the correctness of our model. Since the computation terminates after four steps, the Promela model generated has been capped at ten steps - this is required for formulae with reference to the *step* variable which is always incremented regardless of whether any P system rules were applied, and thus leads to an infinite set of states. This restriction does not undermine the verification process - the P system's computation and step increment are disjoint procedures executed in parallel.

We also note the 1224 byte state vector has been specified to accommodate the compartment data structure, in response to SPIN's requirement for additional memory. For a ten step execution, the verification process did not require more than the default 128.782 MB.

Property	If a node n_i is visited at a particular step, then n_i will acknowledge all its parent nodes by sending a c object.
LTL	$(\text{F } step = 1 \wedge |\langle n_3 \rangle|_a > 0) \rightarrow (\text{F } step = 2 \wedge |\langle n_3 \rangle|_s = 0 \wedge |\langle n_1 \rangle|_c > 0)$
SPIN LTL	`(<> (step == 1 && C[2].x[_a] > 0)) -> (<> (step == 2 && C[2].x[_s] == 0 && C[0].x[_c] > 0))`
Description	The property is verified for a particular instance (node n_3, between steps 1 and 2), however the behaviour may be generalised by verifying the same property for all child nodes in $e\Pi_{dag}$. The left hand side term of the implication identifies the state where n_3 receives an a object from its parent n_1 whilst the right hand side accounts for the reachibility of the following step (step 2) where n_3 has depleted its s object and has sent a c back to its parent compartment. Importantly, this behaviour is not conclusively demonstrated by the clause $|\langle n_1 \rangle|_c > 0$, it may very well be that the multiplicity of c is always greater than 0 due to other nodes sending the cs and not n_3 in particular. It is indeed the case, that the execution of rule $\langle a \rangle : s \longrightarrow c_{\langle Pn_i \rangle}$ is inferred by the absence of object s, whilst a second property (illustrated in Table 7.4) will confirm the presenece of the expected number of cs in every parent compartment.

	Evaluation	Time	Memory	States	State vector
Result	True	0s	128.782 MB	8	1224 bytes

Table 7.2: Property $P_{dag}1$

Property	All child nodes (non-root nodes) will eventually be visited.
LTL	F $step > 0 \wedge \lvert\langle n_1\rangle\rvert_s = 0 \wedge \lvert\langle n_2\rangle\rvert_s = 0 \wedge \lvert\langle n_3\rangle\rvert_s = 0 \wedge \lvert\langle n_4\rangle\rvert_s = 0 \wedge \lvert\langle n_5\rangle\rvert_s = 0 \wedge \lvert\langle n_6\rangle\rvert_s = 0 \wedge \lvert\langle n_7\rangle\rvert_s = 0 \wedge \lvert\langle n_8\rangle\rvert_s = 0 \wedge \lvert\langle n_9\rangle\rvert_s = 0$
SPIN LTL	`<> (step > 0 && C[1].x[_s] == 0 && C[2].x[_s] == 0 && C[3].x[_s] == 0 && C[7].x[_s] == 0 && C[5].x[_s] == 0 && C[8].x[_s] == 0)`
Description	The absence of s objects in a node translates to a visited node who has also sent an acknowledgement to its parent (there is a single s associated with each node and no additional s objects can be produced). The property stipulates that a state where all child nodes have a multiplicity of 0 for object s will eventually be reached.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.782 MB	10	1224 bytes

Table 7.3: Property $P_{dag}2$

Property	The algorithm eventually computes the number of child nodes for each vertex in the DAG.
LTL	F $(\lvert\langle n_1\rangle\rvert_c = 2 \wedge \lvert\langle n_2\rangle\rvert_c = 2 \wedge \lvert\langle n_3\rangle\rvert_c = 2 \wedge \lvert\langle n_4\rangle\rvert_c = 0 \wedge \lvert\langle n_5\rangle\rvert_c = 1 \wedge \lvert\langle n_6\rangle\rvert_c = 1 \wedge \lvert\langle n_7\rangle\rvert_c = 1 \wedge \lvert\langle n_8\rangle\rvert_c = 0 \wedge \lvert\langle n_9\rangle\rvert_c = 1)$
SPIN LTL	`<> (C[0].x[_c] == 2 && C[1].x[_c] == 2 && C[2].x[_c] == 2 && C[3].x[_c] == 1 && C[4].x[_c] == 1 && C[5].x[_c] == 1 && C[7].x[_c] == 0 && C[8].x[_c] == 0)`
Description	The multiplicity of object c in each compartment $i, 1 \leq i \leq N$ will eventually reflect the number of child nodes of i.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.782 MB	10	1224 bytes

Table 7.4: Property $P_{dag}3$

Property	The algorithm will reach the halting configuration - the outcome of the computation - after no more than four steps.
LTL	G $step < 4 \vee (\lvert\langle n_1\rangle\rvert_c = 2 \wedge \lvert\langle n_2\rangle\rvert_c = 2 \wedge \lvert\langle n_3\rangle\rvert_c = 2 \wedge \lvert\langle n_4\rangle\rvert_c = 0 \wedge \lvert\langle n_5\rangle\rvert_c = 1 \wedge \lvert\langle n_6\rangle\rvert_c = 1 \wedge \lvert\langle n_7\rangle\rvert_c = 1 \wedge \lvert\langle n_8\rangle\rvert_c = 0 \wedge \lvert\langle n_9\rangle\rvert_c = 1)$
SPIN LTL	`[] (step < 4 || (C[0].x[_c] == 2 && C[1].x[_c] == 2 && C[2].x[_c] == 2 && C[3].x[_c] == 1 && C[4].x[_c] == 1 && C[5].x[_c] == 1 && C[7].x[_c] == 0 && C[8].x[_c] == 0))`
Description	The property can also be formulated as: it is always the case that the system's configuration (identified as the solution to this problem) is a stable one after the first four steps.

	Evaluation	Time	Memory	States	State vector
Result	True	0s	128.782 MB	25	1224 bytes

Table 7.5: Property $P_{dag}4$

Property $P_{dag}5$ (Table 7.6) is noteworthy since it references a variable $qSum$ which is not part of our model. Similarly to the *step* identifier, $qSum$ is an auxiliary element added to the state vector to store a *computed* or *derived* attribute of interest in $e\Pi_{dag}$. It represents the sum of all q objects across compartments in C and it is calculated using a C function, listed in Fig 7.2. We recall that C fragments in Promela can address the state vector using the (dereferenced) pointer *now*. Thus, `now.C[i].x[_q]` represents the value of x at index $\_q$ for compartment C_i in the state vector, or in simple terms, the multiplicity of q in node n_i. For each state, $qSum$ is computed such that it can be referenced from an LTL formula (as illustrated in $P_{dag}5$).

This technique is practical for buffering computed values which would otherwise be cumbersome or impossible to express in LTL. For instance we cannot apply mathematical functions such as the square root or a derivative with LTL syntax, nor can we include C procedures as part of the temporal logic formulae. Had we not used $qSum$, all nodes would have had to be referenced individually in a rather lengthy summation (C[0].x[_q] + C[1].x[_q] + C[2].x[_q] + C[3].x[_q] + C[4].x[_q] + C[5].x[_q] + C[7].x[_q] + C[8].x[_q]).

We conclude this remark and chapter by stating that an *eps2spin* generated Promela model is not intended to be unalterable, minimal or even optimal for the problem being modelled. On the contrary, the translation is designed to be intelligible, intuitive and easily modifiable, either to include ancillary procedures and variables into the state vector as we have shown in $P_{dag}5$, or to optimise the state transition function, or to isolate and analyse particular segments of the system.

```
void computeQSum() {
    short qSum = 0;
    for(int i = 0; i < now.C_COUNT; ++i) {
        qSum += now.C[i].x[_q];
    }
    now.qSum = qSum;
}
```

Figure 7.2: A C function which computes the sum of all q objects in $e\Pi_{dag}$ for a given system state.

Property	There are precisely two leaf nodes in the DAG structure which underpins $e\Pi_{dag}$.
LTL	G $step < 4 \vee qSum = 2$
SPIN LTL	`[] (step < 4 || qSum == 2)`
Description	Since all nodes are eventually visited and all visited nodes further propagate an a object to their direct descendants consuming the only q available in each compartment, we can identify leaf nodes in $e\Pi$ by searching for compartments which have not applied the $q, a \longrightarrow a_{\langle Cn_i \rangle}$ rule and have consequently retained the object q. More precisely, a compartment containing a q in the halting configuration represents a leaf node in $e\Pi$. Hence, the number of leaf nodes in $e\Pi$ is equal to the sum of q objects across compartments in C - this is computed for each system state and stored in the state vector with the reference `qSum`.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	128.782 MB	25	1224 bytes

Table 7.6: Property $P_{dag}5$

Chapter 8

Solving the Subset Sum problem in linear time

8.1 Objectives

A most fruitful and persistently investigated property of P systems featuring membrane division or creation rules is the ability to increase a computational space exponentially, by executing these rules in parallel. The most prominent application scope of such models is unequivocally the complexity class NP-complete of computational problems. We have enumerated some of the developments based on P system with active membranes in section 3.6 of chapter 3. The referenced case studies present linear time solutions to computationally hard problems and essentially employ the same strategy for generating the computational space required to achieve this performance.

The Subset Sum problem is representative for the NP-complete class because it evinces the underlying necessity to consider *all combinations* of distinct elements of a finite set, in order to produce a result. Consequently, such a problem requires exponential computational resources, either in the temporal (number of computational steps) or spatial (memory) domain, assuming $P \neq NP$.

The Subset Sum problem is stated as follows:

Given a finite set $A = \{a_1, \ldots, a_n\}$, of n elements, where each element a_i has an associated weight, w_i, and a constant $k \in N$, it is requested to determine whether or not there exists a subset $B \subseteq A$ such that $w(B) = k$, where $w(B) = \sum_{a_i \in B} w_i$.

The Subset Sum problem appoints all combinations of integers as subsets of the initial set A, or more accurately, the set of weights respective to elements in

A. It is thus transparent that the number of sums to be evaluated is equal to the cardinality of the power set of A, or, more precisely $2^n - 1$ if we exclude the irrelevant empty set. Since our elements are in fact integers, optimisations have been considered, leveraging the intrinsic order relation between numbers, coupled with efficient sorting algorithms to avoid generating all possible subsets [53]. This did not, however, manage to reduce the complexity of the problem to a non-exponential order.

In this chapter we present a linear time solution to the Subset Sum problem using elementary P systems. A modelling strategy for generating an exponential computational space by parallel membrane instantiation is described, emphasizing how membrane division rules can be expressed with EPS transition rules. In doing so, we demonstrate that elementary P systems are as capable of implementing efficient solutions to NP-complete problems as other P system variants, most notably, P system with active membranes. Moreover, it is shown that model checking is a suitable verification technique for such models due to the fact that the requirement for exponential space (i.e. memory required by state vector) does not entail an exponential number of states for deterministic algorithms. This implies that the traversal of the state space is not undermined by the combinatorial calculation the solution requires.

8.2 EPS Model

We consider the following elementary P system algorithm for solving the Subset Sum problem in linear time. The definition below references n as the cardinality of the set A of integers, $1 \leq i < n$; w_i is weight of the ith element in the set A and k is the constant whose equality to a subset sum is to be determined.

$$e\Pi_{sum} = (O, C, R)$$

where

$O = \{e, Y, N, a, r_i, f, xf, s, p, q\}$;

$C = \{$
$\quad (e, p, q),$
$\quad (p, r_1)$
$\}$;

$R = \{$
$\quad \langle e, !Y, N \rangle : \{$
$\quad\quad q \longrightarrow q, s;$
$\quad\quad \langle s^{n+1} \rangle : p \longrightarrow N;$

```
        }

        $\langle !f \rangle$ : {
            $\langle a^k, !a^{k+1} \rangle : p \longrightarrow f, f_{all}, Y_{\langle e \rangle}$;
            $\langle !a^k \rangle$ : {
                $r_i \longrightarrow r_{i+1}, a^{w_i}, r_{i+1_{new}}, p_{new}$;
                $r_n \longrightarrow a^{w_n}, xf$;
                $\langle !r_n \rangle : a \longrightarrow a, a_{new}$;
            }
            $xf \longrightarrow f$;
        }

    }
```

We initially examine an instance of the Subset Sum problem with $A = \{1, 12, 6, 11, 7, 2\}$, where the weight w of each element in A is given by its integer value (i.e. $w_6 = 6$) and $k = 25$. Expanding the formal model defined above, by iteration over i for $n = 6$ ($i = 1..5$), yields precisely the EPS required for this instance, expressed using the eps modelling language in the following listing.

```
(e, p, q);
(p, r1);

e, !Y, !N: {
    q -> q, s;
    7s: p -> N;
}

!f: {
    25a, !26a: p -> f, [f], [Y @ e];
    !25a: {
        r1 -> r2, a, [r2, p *];
        r2 -> r3, 12a, [r3, p *];
        r3 -> r4, 6a, [r4, p *];
        r4 -> r5, 11a, [r5, p *];
        r5 -> r6, 7a, [r6, p *];
        r6 -> 2a, xf;

        !r6: a -> a, [a *];
    }
    xf -> f;
}
```

To achieve a linear time performance for the NP-complete problem, $e\Pi_{sum}$ implements a parallel expansion of the computational space by means of *membrane instantiation*. We recall that a *new*, quiescent compartment is initiated (activated) when at least one rule with a target indicator *new* is applied in an operational membrane. The compartment triggering the instantiation is not lost (*detached*) as a result of this process and continues to be part of the set C. This contrasts with P systems with active membranes, were membrane division rules 1. duplicate the remaining content of the original compartment σ to the two newly created regions and 2. relinquish σ with its label and additional state elements (such as electric charges).

The model $e\Pi_{sum}$ features two compartments in its initial configuration. The purpose of the compartment containing one e object ($\langle e \rangle$) is to *collect* a 'yes' answer or to *generate* a 'no' answer by the end of the computation. A 'yes' assertion to the investigation is indicated by the Y object, whereas N denotes a negative response. The rule $q \longrightarrow q, s$ increments the multiplicity of object s in $\langle e \rangle$, effectively counting the number of steps. If $n+1$ steps have lapsed and no answer has been computed by this time, then rule $\langle s^{n+1} \rangle : p \longrightarrow N$ generates a N object pronouncing that a subset of A whose weight is equal to the constant k does not exist. Both rules are only applicable in the absence of an answer (Y, N) to the enquiry, as indicated by the scope closure $\langle e, !Y, N \rangle$.

The second compartment is primarily a container of a objects whose multiplicities encode the weights and sum of weights associated to elements from the set A. The $\langle !f \rangle$ top level scope ensures that all rules enveloped only execute if a solution to the problem has yet to be *found*. More precisely, if a compartment is found to contain precisely k objects a ($\langle a^k, !a^{k+1} \rangle$), then an f object is generated and also propagated to all compartments in $e\Pi_{sum}$, signalling the discovery of a sum equal to k and immediately halting the computation. The same rule $p \longrightarrow f, f_{all}, Y_{\langle e \rangle}$ also sends a Y object to the output compartment identified by $\langle e \rangle$.

If the multiplicity of a in a particular compartment σ is strictly lower than k, then one of the rules requiring r_i is executed, instantiating a compartment by sending a p object with target *new* and also adding the *next* weight in the form of a^w objects to σ. Rule $\langle !r_n \rangle : a \longrightarrow a, a_{new}$ replicates the existing a objects in a compartment to a newly instantiated one, if r_n is not present to prohibit this action. In other words, during each computational step (excepting the last), every compartment consisting of a objects will duplicate its content to a new instance and include a number of as that is equal to the weight of the next element in A. This expansion is illustrated in Figure 8.1 for the first three computational steps of $e\Pi_{sum}$. This depiction does not reflect a complete configuration at any given step but rather focuses on the *accelerated* computation attained by the parallel instantiation of EPS compartments (operational regions).

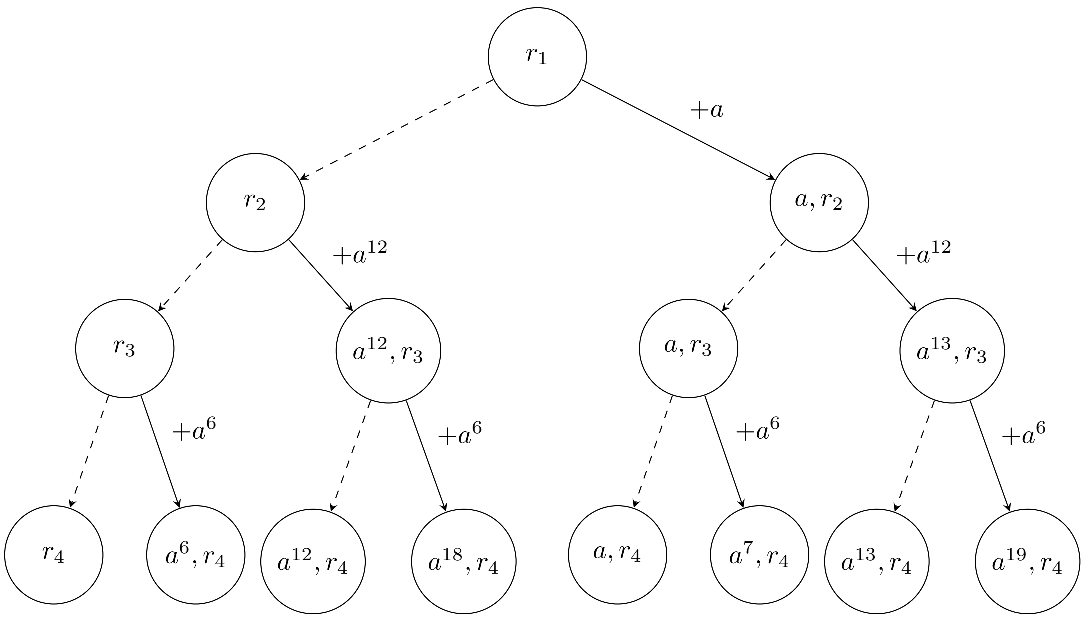

Figure 8.1: Generating an exponential computational space using elementary P systems.

In this respect, we acknowledge the absence of object p in each circle representing a compartment and the transitions observed in $\langle e \rangle$ - the output compartment. We also highlight the fact that a single compartment is created per step - indicated by the dashed arrow, whilst the persistence of 'initiator' membranes is symbolised by plain arrows.

We further stress the importance of rule $a \longrightarrow a, a_{new}$ which identifies an operational pattern with frequent recurrence when such expansive dynamic is exercised in EPS. In order to equate the semantic significance of a membrane division rule (emblematic to P systems with active membranes), it is necessary to replicate the content of an EPS compartment to its quiescent counterpart. This is achieved by applying rules of the form $x \longrightarrow x, x_{new}$ for each x object in $O' \subseteq O$ in a maximally parallel manner. Hence, not only can an EPS simulate membrane division, but also it provides for more controlled replication of objects at the expense of additional rewriting rules.

$e\Pi_{sum}$ reaches a halting configuration when

1. all compartments bearing a objects also include an f and
2. $\langle e \rangle$ contains *at least* one Y or N.

If a sum equal to k is not found, then each compartment produces an f by executing the rule $r_n \longrightarrow a^{w_n}, f$, preventing any further rule application. The result of the computation is construed from the presence of Y or N objects in $\langle e \rangle$ which are mutually exclusive.

Remark 8.1. The EPS algorithm embodied by $e\Pi_{sum}$ will reach a halting configuration after a maximum $n + 2$ steps, where n is the size of the instance being modelled, that is the cardinality of the set of elements A.

Remark 8.2. The algorithm requires no more than $2^{n-1} + 1$ compartments to generate an answer to the Subset Sum problem. If a solution is found, i.e. a subset whose sum is equal to k, then the number of compartments utilised is *maximum* $2^{step-1} + 1$, where *step* represents the computational step the solution was found at. It may be the case fewer compartments are generated since computation on invalid paths, that is paths with $|a| > k$, is immediately pruned.

Remark 8.3. The number of Y objects in the output compartment identified by $\langle e \rangle$ is equal to the number of subsets whose sum is k found *during the same computational step*. It is possible that more such subsets can be identified, however, $e\Pi_{sum}$ will halt after the first subset is encountered. Since each subset sum is computed in parallel, it may be the case that multiple k values are identified during the same step, yielding more than one Y in $\langle e \rangle$.

8.3 Verification results

The following properties affirm the validity of our EPS based algorithm in a formal manner. All statements evaluate to true when verified against the translated Promela model. We note that the `eps2spin` conversion tool cannot infer the maximum number of compartments required by the model until a halting configuration is reached, nor can it determine whether the P system is halting or not. Since SPIN requires a fixed size state vector (it cannot dynamically allocate memory for newly generated elements), it is important that the user specifies the `MAX_COMPARTMENT_COUNT` and `MAX_STEPS` constants via arguments to the `eps2spin` command:

```
node eps2spin SubsetSumYes.eps -o SubsetSumYes.pml -n 10 -x 129
```

where `n` is the number of steps we restrict the model to and `x` is the maximum number of compartments the system may require.

Since it is known that a maximum of $n + 2$ steps are required to compute an answer to problem, it is sufficient to limit the number of steps of our model (an instance with $n = 6$) to ten (or any number greater than eight). We remark that the state vector is of constant size (12424 bytes) for all experiments, a requirement that was accommodated with the SPIN directive `DVECTORSZ=12500`. Additionally, the memory utilised by the model checker is always 131.576 MB, the base value pre-computed by SPIN based on the hash table structures employed and the state vector.

We also note that the total number of states SPIN reports is not precisely equal to the number of steps of $e\Pi_{sum}$ which is the ideal scenario for purely deterministic algorithms. The translated model always consists of auxiliary states due to the limitations of SPIN on *hiding* - that is marking certain states as irrelevant and excluding these from the search process. For instance, it is impossible to specify an initialised complex data model such as the one an elementary P system is projected to, in the initial state of the system. To overcome this shortcoming, the step variable is initialised as -1 and only becomes 0 after the model has been initialised with the respective values reflecting the initial configuration of the EPS. It is always the case, however, that one state is superfluously included in state graph, which represents the *pre-initialisation stage*, step -1.

Whilst there are other constructs which inevitably *pollute* the model's generated state space, it is important to note that *at most a linear* increase is observed as a result of the translation between the two specifications (EPS and Promela) and the ancillary states introduced. This is vital to our approach as it determines the feasibility of model checking such systems.

We also include the experimental results of three extended instances of the Subset Sum problem in Table 8.3. Property $P_{sum}2$ was successfully verified on these models and whilst an increase in memory for models with a greater cardinality of A is notable, the time reported by SPIN remains insignificant. It was, however, noticed that SPIN does take more time to perform a pre-search initialisation and this varies considerably, based on the state vector size. This is not reported by the tool as part of the verification process.

Property	If there is at least one compartment containing k number of as, then a 'yes' answer is computed.
LTL	$(\text{F } card(\langle a^k \rangle) > 0) \rightarrow (\text{F } \lvert\langle e \rangle\rvert_Y > 0)$
SPIN LTL	`<> (kIndex > -1) -> <> (C[0].x[_Y] > 0)`
Description	The SPIN LTL formula employs a variable `kIndex` to store the index of the first compartment which features precisely k objects a. If such an index is greater than -1, then the designated output compartment $\langle e \rangle$ must receive a Y object indicating the affirmative response to the decision problem. The procedure which computes `kIndex` is listed in Fig 8.2. In the LTL formula above, *card* denotes the cardinality of the set of compartments congruent with scope $\langle a^k \rangle$, that is the number of compartments with precisely k objects a.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	16	12424 bytes

Table 8.1: Property $P_{sum}1$

```
void findK() {
    now.kIndex = -1;
    for(short i = 0; i < now.C_COUNT; ++i) {
        if(now.C[i].x[_a] == 25) {
            now.kIndex = i;
            break;
        }
    }
}
```

Figure 8.2: A C function which searches for a compartment containing precisely $k = 25$ objects a. The index is stored in the state vector. The negative index -1 corresponds to the case where no such compartment can be found.

Property	If a 'yes' answer is computed then, there is at least one compartment having the multiplicity of a equal to k.
LTL	G $(\lvert\langle e\rangle\rvert_Y > 0 \rightarrow card(\langle a^k\rangle) > 0)$
SPIN LTL	`[] (C[0].x[_Y] -> kIndex > -1)`
Description	The property represents the complementary implication of $P_{sum}1$. The combination of the two formulae demonstrate that the algorithm computes an affirmative answer if and only if there exists at least one compartment whose multiplicity of a objects is equal to k.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	25	12424 bytes

Table 8.2: Property $P_{sum}2$

Set A	k	States	Time	Memory	State vector
$A_1 = \{5, 8, 7, 6, 9, 0, 1\}$	36	18	0s	133.722 MB	11240 bytes
$A_2 = \{3, 8, 18, 25, 12, 17, 22, 9, 10\}$	124	22	0.06s	217.407 MB	62040 bytes
$A_3 = A_2 \cup \{39, 42, 30, 44, 45, 40\}$	70	45	0.03s	184.744 MB	84048 bytes

Table 8.3: Verification of $P_{sum}2$ for various configurations (A, k).

Property	At least one solution to the problem will be observed after precisely 6 computational steps.
LTL	F $step = 6 \wedge \lvert\langle e\rangle\rvert_Y > 0$
SPIN LTL	`<> (step == 6 && C[0].x[_Y] > 0)`
Description	The number of solutions identified during a step equates to the multiplicity of Y present in $\langle e\rangle$. Thus, it takes five steps to generate a subset whose sum is $k = 25$ and an additional step to halt computation and emit the Y object indicative of an affirmative response.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	14	12424 bytes

Table 8.4: Property $P_{sum}3$

Property	It takes at least six steps to compute a solution to the problem.
LTL	$\lvert\langle e\rangle\rvert_Y = 0$ U $step = 6 \wedge \lvert\langle e\rangle\rvert_Y > 0$
SPIN LTL	`C[0].x[_Y] == 0 U (step == 6 && C[0].x[_Y] > 0)`
Description	Whilst the preceding property (Table 8.4) verifies the existence of a solution after six steps, the formulae expressed above stipulates that no solution can be found during the first six computational steps of $e\Pi_{sum}$.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	14	12424 bytes

Table 8.5: Property $P_{sum}4$

Property	A ‘no’ answer to the decision problem is never computed.
LTL	G $\lvert\langle e\rangle\rvert_N = 0$
SPIN LTL	`[] (C[0].x[_N] == 0)`

Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	14	12424 bytes

Table 8.6: Property $P_{sum}5$

Property	No more than 31 compartments are required for this particular instance to reach a halting configuration.
LTL	G $compartmentCount \leq 31$
SPIN LTL	`[] (C_COUNT <= 31)`
Description	The `C_COUNT` variable represents the number of compartments at a certain step and is part of the state vector. When compartment instantiation is employed, `C_COUNT` will increase to reflect the expansion.

Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	25	12424 bytes

Table 8.7: Property $P_{sum}6$

Property	The number of compartments utilised is equal to $2^{step} + 1$, for $0 \leq step < n - 2$ in this model.
LTL	G $step > 4 \vee compartmentCount = stepExp$
SPIN LTL	`[] (step > 4 || C_COUNT == stepExp)`
Description	The LTL formula makes reference to an exponential expression of *step* which is computed by an ancillary C procedure (Listing 8.3). The validity of this property demonstrates the exponential growth of the computational space achieved by EPS compartment instantiation. This growth is reduced at step four when two compartments of $e\Pi_{sum}$ have generated values (weight sums) that are greater than $k = 25$, preventing subsequent instantiation on the respective paths. Consequently the total number of compartments required for a halting configuration is less than $2^{step-1} + 1 = 33$ - a statement proven by $P_{sum}6$ (Table 8.7).

Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	25	12424 bytes

Table 8.8: Property $P_{sum}7$

```
void stepExp() {
    now.stepExp = 1;
    for(short i = 0; i < now.step; ++i) {
        now.stepExp *= 2;
    }
    now.stepExp += 1;
}
```

Figure 8.3: An auxiliary C function which computes the expression $2^{step} + 1$ for $e\Pi_{sum}$.

We now consider a second instance to the problem, having the same set of elements $A = \{1, 12, 6, 11, 7, 2\}$, however the constant k is required to be 35. As we will see in the outcome of our verification, there is no subset $B \subseteq A$ such that $w(B) = 35$.

The eps model pertinent to this scenario is almost identical to the one presented earlier, since r_i rules generate the same number of as at each step and the cardinality of A remains $n = 6$. The only adjustment required is the scope and scope closure which refer to the constant k. We recall that elementary P systems do not provide any means for sequential substitutions or pre-processing, rather they implement a pure maximally parallel transition paradigm. Consequently, the value of k must be explicitly stated as a multiplicity and cannot be used as a variable in a scope construct.

```
(e, p, q);
(p, r1);

e, !Y, !N: {
    q -> q, n;
    7n: p -> N;
}

!f: {
    35a, !36a: p -> f, [f], [Y @ e];
    !35a: {
        r1 -> r2, a, [r2, p *];
        r2 -> r3, 12a, [r3, p *];
        r3 -> r4, 6a, [r4, p *];
        r4 -> r5, 11a, [r5, p *];
        r5 -> r6, 7a, [r6, p *];
        r6 -> 2a, xf;

        !r6: a -> a, [a *];
    }
    xf -> f;
}
```

Evidently, the absence of an internal iterative expansion in EPS does not entail that various instances of the problem must be devised manually. On the contrary, most examples presented in this thesis were subject to numerous experiments in various configuration and as such, *model generators* were the preferred and much more efficient technique to create the systems. A model generator is simply a *linear parametric expansion* of the formal definition of the system. Its role is to perform the symbol substitutions in objects expressed as indexed identifiers or static references (such as r_i, a^k etc). This transformation is effectuated once only and is irrelevant to the P system model and its functioning; it is a method of convenience since it allows us to express an EPS more concisely.

We initially verify the very first two properties described earlier, $P_{sum}1$ and $P_{sum}2$ which must hold regardless of the answer eventually computed for the Subset Sum problem. Whilst their validity was proved by SPIN in the same insignificant time lapse (under one second), the number of states required for the first property is 25, in contrast to 16 reported for the 'yes' instance. This is consistent with our expectation, since the 'finally' (F) statements do not descend to the same depth when a Y object is found and $kIndex > -1$; the formula is deemed valid at this point. Conversely, if $kIndex$ is never assigned a value greater than -1, then the complete state space for $e\Pi_{sum}$ must be generated and traversed[1].

Property	If a compartment having the multiplicity of a equal to k is never found, then a 'no' answer is eventually generated.
LTL	(G $kIndex = -1$) $\rightarrow$ (F $\lvert\langle e\rangle\rvert_N > 0$)
SPIN LTL	`([] (kIndex == -1)) -> (<> (C[0].x[_N] > 0))`
Description	The 'no' answer to the decision problem is denoted by the presence of an N object in the output compartment $\langle e\rangle$. Since it is this compartment only which generates N, the multiplicity of N can be maximum one. This contrasts with the number Y objects which can be present in $\langle e\rangle$, reflecting the number of subsets found at a given step to contain k objects a.

	Evaluation	Time	Memory	States	State vector
Result	True	0s	131.576 MB	18	12424 bytes

Table 8.9: Property $P_{sum}8$

[1]We underline SPIN's advantage of generating the state space 'on the fly'; LTL formulae using the *finally* operator (F) may only require a subset of states the model exhibits. The search remains, of course, exhaustive for this sector of the computation, that is the collection of states pertinent to the verification.

Property	The instance does not yield a solution to the problem and consequently, a 'no' answer is computed after $n+2$ (8) steps.				
LTL	F $step = 8 \wedge |\langle e \rangle|_N > 0$				
SPIN LTL	`<> (step == 8 && C[0].x[_N] > 0)`				
Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	18	12424 bytes

Table 8.10: Property $P_{sum}9$

Property	It takes a minimum of $n+2$ (8) steps to compute a 'no' answer to the problem.				
LTL	$|\langle e \rangle|_N = 0$ U $(step = 8 \wedge |\langle e \rangle|_N > 0)$				
SPIN LTL	`C[0].x[_N] == 0 U (step == 8 && C[0].x[_N] > 0)`				
Description	The property asserts that not only is a 'no' answer computed, but also such an answer can only be generated after eight steps, importantly, after all combinations have been verified in time that is linear to the size of the problem $n = 6$ (there are an additional two steps required to reach the halting configuration).				
Result	Evaluation	Time	Memory	States	State vector
	True	0s	131.576 MB	18	12424 bytes

Table 8.11: Property $P_{sum}10$

Property	A 'yes' answer is never computed.
LTL	G $\lvert\langle e\rangle\rvert_Y = 0$
SPIN LTL	`[] (C[0].x[_Y] == 0)`
Description	In conjunction with $P_{sum}5$, this property validates the claim that 'yes' and 'no' answers are mutually exclusive, *for the two models investigated.* Notably, the formula does not entail that the P system cannot by construction generate both a Y and an N in one computation, which is of course the case with $e\Pi_{sum}$. Since we are verifying two different instances of a decision problem (whose answers can either be 'yes' and 'no' and are immutable), it is impossible to formally demonstrate this claim *for all possible instances* of the Subset Sum problem using model checking.

	Evaluation	Time	Memory	States	State vector
Result	True	0s	131.576 MB	25	12424 bytes

Table 8.12: Property $P_{sum}11$

Property	No more than 33 compartments are required to reach a halting configuration.
LTL	G compartmentCount ≤ 33
SPIN LTL	`[] (C_COUNT <= 33)`
Description	Since $k = 35$ is much higher than for the previous scenario (25), the sum of weights will not exceed this value until after the computational space has expanded to $2^n + 1$ compartments, which is the maximum required by a computation of $e\Pi_{sum}$.

	Evaluation	Time	Memory	States	State vector
Result	True	0s	131.576 MB	25	12424 bytes

Table 8.13: Property $P_{sum}12$

Chapter 9

The Dining Philosophers problem

9.1 Objectives

P systems are inherently distributed models because they exercise multiset transition rules across multiple compartments - clearly delineated sectors of computational space. The term *membrane* (in *membrane computing*) itself implies a permeable separation between two distinct regions where transformations can occur independently. Such systems can also be described as concurrent since there is no time unit distinction between (perceptibly) individual transitions taking place within and across disjoint computational spaces.

One of the compelling topics which circumscribes concurrent computation is the problem of *synchronisation* and, strictly related, *starvation.* Ensuring the *correct and continuous* functioning of a distributed system is a matter of decisive significance which must be primarily addressed. More specifically, a requirement to avoid a deadlock, a state where two or more processes compete for resources but cannot advance since these are mutually held, is intransigent for systems with distributed, communicating nodes. This requirement has been captured and is eloquently evinced in the Dining Philosophers problem, originally introduced in 1965 by Edsger Dijkstra and subsequently adapted to its present formulation by C. A. R. Hoare [51]. The problem is described as follows:

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can of course be used by one philosopher only at a time. After he finishes eating, he must put down both forks such that they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before acquiring both. It is required to design a concurrent algorithm which excludes any possibility of *reaching a deadlock.*

The problem often specifies an additional requirement, commonly referred to as starvation avoidance or inclusion of a *fairness* property. This stipulates that no philosopher can continuously consume spaghetti such that his neighbours at the table are permanently deprived of at least one of the necessary forks. This is especially important when considering an infinite supply of spaghetti and hence, a philosopher may be perpetually blocked from eating.

Formal verification is paramount in asserting the correctness of protocols for distributed models or frameworks regulating the dynamics of systems which entail a plurality of entities. In particular, model checking is one of the principal techniques for validating systems whose process interaction may lead to unacceptable states, such as a deadlock.

In this chapter we present an innovative EPS based solution to the Dining Philosophers problem. Our modelling approach diverges from traditional strategies by considering 'personalities' or pre-established (static) behavioural patterns, as opposed to using a mediator (arbiter) or any form of sequential regulator (queuing scheme). On the one hand, the algorithm requires no central authority after initialisation, it is purely distributed and parallel, and on the other hand, it remains faithful to the problem requirements since philosophers do not communicate to achieve their goal. Furthermore, it is formally demonstrated by SPIN model checking that one distinction in the behavioural pattern of N philosophers with finite units of spaghetti is sufficient to avoid a deadlock and implicitly, starvation.

9.2 Model

The design of our EPS model for the Dining Philosophers problem assumes the following ontological statements:

1. Two entities with different coordinates in at least one dimension (i.e. space, time etc) are distinct and coexist in a plurality.
2. The dynamics of two distinct entities can never be equivalent outside of a restrictive context. If such a context is considered, then there exists an outer context where the distinction is actual and can be exerted into an inner context.

The two highly general affirmations insist that there is an inherent variation in a plurality, or if a limited perspective is considered where any distinctive influences on the dynamics of its entities are not discernible, then dissimilarities can be introduced from an external context. If we are to consider $N > 1$ philosophers, then, by the very fact they coexist in a plurality it is entailed there are aspects which influence their behaviour such that no two philosophers act identically. If we project the N philosophers into a scientific context and consider the same abstract model for each individual, then we are also able to 'inject' a property to one or more philosophers that is distinct within the group.

The following definition represents an elementary P system model of N dining philosophers distributed and operating as specified above:

$$e\Pi_{phil} = \{O, C, R\}$$

where, for an instance $phil_N$ of the problem with $N \geq 3$ philosophers each having a bowl of 3 spaghetti units and a patience value of 3; N forks, $1 \leq i \leq N$ and $j = (N - 2 + i) \mod N + 1$:

```
$O = \{Phil_i, F_i, s, f, f_i, p, q, rP_i, z\}$
$C = \{(Phil_i, p^3, s^3, q)_i, (F_i, f)_i\}$
$R = \{$

  $\langle Phil_i \rangle : \{$
    $f_i, f_j, s \longrightarrow f_{\langle F_i \rangle}, f_{\langle F_j \rangle};$
    $\langle s \rangle : \{$
      $\langle !f_i | !f_j \rangle : \{$
        $q, p \longrightarrow q, rP_{i\ \langle F_i \rangle}, rP_{i\ \langle F_j \rangle};$
        $\langle !p \rangle : \{$
          $f_i \longrightarrow f_{\langle F_i \rangle};$
          $f_j \longrightarrow f_{\langle F_j \rangle};$
          $q \longrightarrow q, p^3;$
        $\}$
      $\}$
    $\}$
  $\}$


  $\langle F_i \rangle : \{$
    $f, rP_i \longrightarrow f_{i\ \langle Phil_i \rangle};$
    $f, rP_j \longrightarrow f_{i\ \langle Phil_j \rangle};$
    $\langle !f \rangle : \{$
      $rP_i \longrightarrow z;$
      $rP_j \longrightarrow z;$
```

```
            }
          }

        }
```

Each philosopher i is identified by a compartment with precisely one object $Phil_i$ whilst each fork location on the table is represented by a compartment congruent with the scope $\langle F_i \rangle$, that is, consisting of one object F_i. The behaviour of a philosopher, expressed by the set of rules within the scope closure $\langle Phil_i \rangle$, can be summarised as follows: If $Phil_i$ has remaining spaghetti units and 1. has acquired both his adjacent forks, he will immediately consume one spaghetti unit (denoted by one object s) and return the two forks to their respective positions. Otherwise, if 2. a single or no fork has been acquired, then attempt to pick up the missing fork(s). Each philosopher may continue attempting to acquire the necessary forks as long as his *patience* allows it. When a philosopher runs out of patience waiting for the missing fork(s) to become available, then he will return the fork held in his possession if any, and shall re-start the process of acquiring both forks with 'renewed' patience. This repetitive pattern is followed until the philosopher has exhausted all units of spaghetti in his bowl.

The patience of a philosopher, denoted by the object p in our model, constitutes a personality trait and it is the one configurable element during a supposed initialisation stage. For each attempt to acquire a necessary fork, the multiplicity of p is decremented (i.e. the philosophers patience is gradually lost) and eventually, if there is one fork in his possession, it is returned to its original location. This anthropomorphic behaviour ensures that resources (i.e. forks) are never held indefinitely and always restored to their original state (forks are returned to their respective $\langle F_i \rangle$). Consequently, starvation is avoided by behavioural complementarity and not due to a governing entity which arbitrates the distribution of resources. As we will see in section 9.4, a deadlock can only occur in a sterile environment where all philosophers exhibit the same personality trait (that is, the same patience value) and may block each other indefinitely.

Since any two neighbouring philosophers compete for the same fork, it is nondeterministically decided who is the next recipient of a fork f at F_i, between $Phil_i$ and $Phil_{i+1}$ (for $1 \leq i < N$). This arbitrary 'allocation' is effected in the $\langle F_i \rangle$ compartments, which also discard any requests/attempts that can't be fulfilled (because the fork has already been acquired). Since elementary P systems cannot destroy objects, a rule of the form $rP_i \longrightarrow z$ translates a request object rP_i to an auxiliary z, such that no failed attempt to acquire the fork persists between two consecutive steps. The multiplicity of object z thus represents the number of unsatisfied requests for a particular fork.

It is important a philosopher always returns one fork to his left and the other

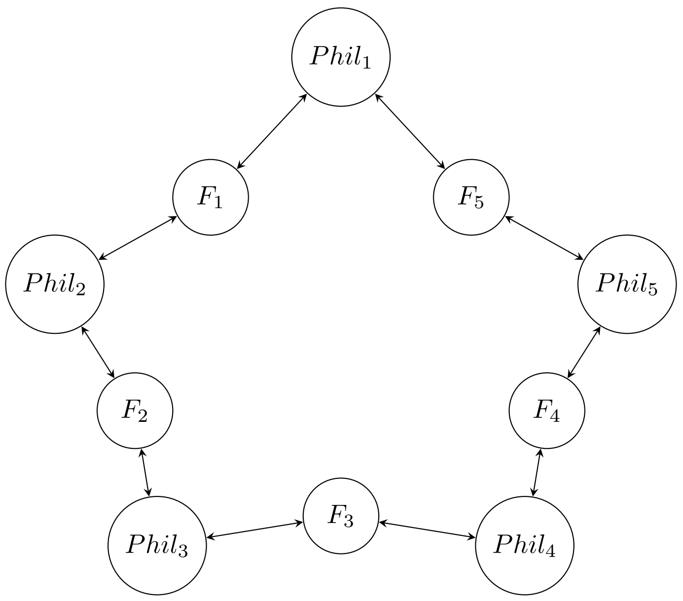

Figure 9.1: Depiction of $e\Pi_{phil}$ for an instance $phil_5$ of the dining philosophers problem

to his right hand side, once he has finished eating or if he runs out of patience. To preserve this consistency, a fork object (f) always bears the index of its location (compartment F_i) when delivered to a philosopher: $f, rP_i \longrightarrow f_i\ _{\langle Phil_i \rangle}$. The philosopher can then return f_i to its corresponding compartment F_i, where the index is not relevant: $f_i \longrightarrow f_{\langle F_i \rangle}$. The distribution of philosophers and forks on an envisaged round table is depicted in figure 9.1. The bi-directional arrows indicate that EPS communication rules are only applied between a philosopher compartment and the two fork compartments in front of it; there is no interaction between any two philosophers at the table.

The following listing represents the eps code derived by expanding the formal model definition for an instance of the problem with $N = 5$ philosophers and forks. The number of spaghetti units and patience value has been retained as 3 for all philosophers, although this can be configured as well. A compact rendition consisting of compartments $Phil_1$ and F_1 is provided below, whilst a complete listing is available in appendix D.

```
(Phil1, 3p, 3s, q);
(F1, f);

Phil1: {
  f1, f5, s -> [f @ F1], [f @ F5];
  s: {
    !f1 | !f5: {
      q, p -> q, [rP1 @ F1], [rP1 @ F5];
      !p: {
          f1 -> [f @ F1];
          f5 -> [f @ F5];
          q -> q, 3p;
      }
    }
  }
}

F1: {
  f, rP1 -> [f1 @ Phil1];
  f, rP2 -> [f1 @ Phil2];
  !f: {
    rP1 -> z;
    rP2 -> z;
  }
}
```

We also illustrate the first ten computational steps (Fig 9.1 and 9.2), as output by our eps simulator, of $e\Pi_{phil}$ in the aforementioned configuration. For conciseness, only mutable objects of the multisets in $\langle Phil_i \rangle$ and $\langle F_i \rangle$ have been included in the two listings. Objects $Phil_i$ and F_i, $1 \leq i \leq N$ have been omitted, since their presence in the respective compartments is permanent. Nevertheless, all objects across all system compartments (the set C) collectively constitute the configuration of $e\Pi_{phil}$ at a particular step.

9.3 Extensions

The model exemplified in the previous section is a 'minimal' representation of our proposed algorithm. Whilst it portrays a paradigm whereby the behavioural pattern of a philosopher, which can be individualised by specifying (pre-establishing) a patience value, is decisive for avoiding starvation, there are various considerations which can be accommodated by simple extension of our basic model.

Suppose we wish to specify that an arbitrary number of spaghetti units is to be consumed by a philosopher when he acquires both forks. A rule $\langle !t^5 \rangle : f_i, f_j, s \longrightarrow t$ could be included in addition to the existing $f_i, f_j, s \longrightarrow f_{\langle F_i \rangle}, f_{\langle F_j \rangle}$. The former will expend one s object (spaghetti unit) without returning the forks, as long as the maximum number of spaghetti units he can consume in succession is not greater than the multiplicity of t (5), which can also be included as a configurable

Step/Scope	$\langle Phil_1 \rangle$	$\langle Phil_2 \rangle$	$\langle Phil_3 \rangle$	$\langle Phil_4 \rangle$	$\langle Phil_5 \rangle$
0	p^3, q, s^3	p^3, q, s^3	p^3, q, s^3	p^3, q, s^3	p^3, q, s^3
1	p^2, q, s^3	p^2, q, s^3	p^2, q, s^3	p^2, q, s^3	p^2, q, s^3
2	f_5, p, q, s^3	f_1, f_5, p, q, s^3	p, q, s^3	f_3, f_4, p, q, s^3	p, q, s^3
3	f_5, q, s^3	p, q, s^2	q, s^3	p, q, s^2	q, s^3
4	f_1, p^3, q, s^3	q, s^2	f_2, f_3, p^3, q, s^3	q, s^2	f_4, p^3, q, s^3
5	f_1, p^2, q, s^3	p^3, q, s^2	p^3, q, s^2	p^3, q, s^2	f_4, p^2, q, s^3
6	f_1, p, q, s^3	p^2, q, s^2	p^2, q, s^2	p^2, q, s^2	f_4, f_5, p, q, s^3
7	f_1, q, s^3	f_2, p, q, s^2	f_3, p, q, s^2	p, q, s^2	p, q, s^2
8	f_5, p^3, q, s^3	f_2, q, s^2	f_3, q, s^2	f_4, q, s^2	q, s^2
9	f_5, p^2, q, s^3	f_1, p^3, q, s^2	p^3, q, s^2	p^3, q, s^2	p^3, q, s^2
10	f_5, p, q, s^3	f_1, p^2, q, s^2	p^2, q, s^2	p^2, q, s^2	p^2, q, s^2

Table 9.1: Ten step simulation of $e\Pi_{phil}$ with $N = 5$ and $p(i) = 3$, $1 \leq i \leq N$ - Philosopher compartments

personality attribute. Since a non-deterministic choice occurs between the two rules, the resulting behaviour is that one or more, up to a maximum five spaghetti units can be arbitrarily consumed during one step[1].

Another noteworthy adjustment is the inclusion of a *refractory period* after patience has been exhausted or spaghetti has been eaten. This would account for the so called *thinking time* as indicated by the problem statement and can be included as a w object with pre-defined multiplicity. A scope closure $\langle !w \rangle$ is used to circumscribe the present rules, such that a philosopher is completely neutral and does not initiate any actions when he is in a *thinking state*. Each step, the multiplicity of w decrements by one, simulating a time lapse of $|Phil_i|_w$ units: $q, w \longrightarrow q$. The rule employed to replenish the patience of a philosopher will also generate the h thinking time units associated: $q \longrightarrow q, p^3, w^h$. Additionally, such a technique can be utilised between subsequent attempts to acquire the fork(s) to account for other factors of randomness. Moreover, the patience property could also be altered based on fork acquisition. For instance a philosopher could

[1]The multiplicity of t can be reset by an ancillary rule which executes when the forks have been returned. Further dynamics can be infused into the system, by removing arbitrary t objects each time the forks are returned.

Step/Scope	$\langle F_1\rangle$	$\langle F_2\rangle$	$\langle F_3\rangle$	$\langle F_4\rangle$	$\langle F_5\rangle$
0	f	f	f	f	f
1	f, rP_1, rP_2	f, rP_2, rP_3	f, rP_3, rP_4	f, rP_4, rP_5	f, rP_5, rP_1
2	rP_1^2, rP_2	rP_2, rP_3^2	rP_3^2, rP_4	rP_4, rP_5^2	rP_5^2, rP_1
3	f, rP_1, z^3	f, rP_3, z^3	f, rP_3, z^3	f, rP_5, z^3	rP_5, rP_1, z^3
4	rP_2, z^3	rP_2, z^3	rP_4, z^3	rP_4, z^3	f, z^5
5	rP_1, z^4	f, z^4	f, z^4	rP_5, z^4	f, rP_1, rP_5, z^5
6	rP_1, rP_2, z^5	f, rP_2, rP_3, z^4	f, rP_3, rP_4, z^4	rP_4, rP_5, z^5	rP_1^2, rP_5, z^5
7	rP_1, rP_2, z^7	rP_2, rP_3^2, z^4	f, rP_3, rP_4^2, z^4	f, rP_4, z^7	f, rP_1, z^8
8	f, rP_2, z^9	rP_2, rP_3, z^7	rP_3, rP_4, z^7	rP_5, rP_4, z^7	rP_5, z^8
9	rP_1, z^9	f, z^9	f, z^9	f, z^9	rP_1, z^9
10	rP_1, rP_2, z^{10}	f, rP_2, rP_3, z^9	f, rP_3, rP_4, z^9	f, rP_4, rP_5, z^9	rP_1, rP_5, z^{10}

Table 9.2: Ten step simulation of $e\Pi_{phil}$ with $N = 5$ and $p(i) = 3$, $1 \leq i \leq N$ - Fork compartments

gain m patience after he had successfully retrieved both forks. Conversely, each successive return of the fork in possession would replenish an increased or reduced number of p objects.

Importantly, these inclusions do not introduce the risk of a deadlock unless, collectively, they allow *all* philosophers at the table to operate *identically, indefinitely.*

Whilst traditional P systems employ finite multisets of objects in transition and communication rules, one may consider an infinite supply of spaghetti by continuous mutation (multiplication) of s objects. More precisely, during each step, $x \geq t$ spaghetti units are added to a philosopher's bowl, where t is the maximum number of spaghetti units that can be consumed per step. To avoid starvation in this particular scenario and perhaps introduce a fairness constrain, it is required to convert the $\langle F_i\rangle$ compartments into 'stateful' components. A fs_i object can be generated each time the fork at F_i has been sent to philosopher $Phil_i$. After a number of consecutive allocations y, it is enforced that the fork will be deterministically assigned to $Phil_j$ instead, if both a rP_j and rP_i objects are present: $\langle rP_i, rP_j, !fs_i^y\rangle : f, rP_i \longrightarrow fs_i, L_i, f_i\ {}_{\langle Phil_i\rangle}$ and $\langle rP_i, rP_j, !fs_j^y\rangle : f, rP_j \longrightarrow fs_j, L_j, f_i\ {}_{\langle Phil_j\rangle}$. Whenever a fs_i marker is generated, the multiplicity of fs_j can be reset since the $Phil_j$ allocation sequence (if there was one) had been broken, and vice-versa: $\langle L_i \rangle : \{fs_j \longrightarrow z;\ L_i \longrightarrow z\}$ and $\langle L_j \rangle : \{fs_i \longrightarrow z;\ L_j \longrightarrow z\}$, where L_i denotes the last philosopher $Phil_i$ the fork was acquired by and z is simply utilised as a *nil* like object, irrelevant to the subsequent computation[1]. It is therefore the case that no philosopher is allowed to acquire the fork for more than y consecutive times, *when requested by two philosophers simultaneously.* The number can exceed y if the prospective contender is in a thinking state for more than the time required to acquire the fork $y' > y$ times by his neighbour.

An alternative strategy which compels equilibrium is to simply grant the fork to the philosopher whose number of requests (attempts to acquire it) is greater. This entails that rP_i, rP_j objects are never discarded, but instead their difference determines the next fork recipient. Since philosophers in $e\Pi_{phil}$ do not continuously request the fork when they consume spaghetti units, the ratio rP_i/rP_j can not constantly equate to one.

Evidently, the fork compartments $\langle F_i \rangle$ are now associated with logic which infringes the neutrality they originally aspired to. Indeed, the fork compartments can be regarded as arbiters which keep a record of who the fork was last assigned to and how many times, but more importantly these compartments implement a procedure to *determine* who the next fork recipient should be. By contrast, in our initial approach, $\langle F_i \rangle$ compartments are state-less and have no functional significance other than to house a non-deterministic selection between two competing requests. The compartment is used to neutrally denote the fork location and has no functional logic ascribed to it.

9.4 Verification results

We begin this section by remarking that although $e\Pi_{fib}$ does not guarantee a halting configuration - the philosophers may repeat the same actions each step and receive a single fork only by non-deterministic choice, with infinite recurrence - the system has nevertheless a finite number of states. Consequently, no ancillary provisions are required by the model to restrict the computation in any way (such as limiting the number of steps), the primary requirement for SPIN model checking is satisfied.

Whilst the standard Promela translation reflects this property outright, a simple adjustment to the five fork related procedures results in a significant op-

[1]The accompanying rules may require amendments as well, to ensure this procedure is not enforced unnecessarily, that is if the philosophers do not contend for forks, then the fs_i and L_i objects are ignored and the request is satisfied accordingly, still tracked however by generating a fs_i object.

timisation. We recall that elementary P systems cannot destroy objects and consequently a rule of the form $rP_i \longrightarrow z$ is employed to dispose of requests rP_i. If philosophers eventually exhaust their spaghetti (i.e. a deadlock does not occur), then the production of z objects is also eventually halted. The detrimental consequence is, however, the greatly increased number of states SPIN must generate due to the variable multiplicity of z. For each incremental update of $|\langle Phil_i \rangle|_z$, a new state is created, whereas, in the absence of this auxiliary object, an existing state would be revisited. In other words, rules of type $rP_i \longrightarrow z$ introduce unnecessary disparity between system states which in turn significantly increases the model checker's search space.

The issue has been addressed by eliminating the creation of z objects from the Promela model. Rather than cumulating the number of failed attempts to pick up a fork, each request object is discarded by assigning a zero multiplicity directly, in the C procedure: `now.C[cIndex].x[_rP1] = 0;`.

Property	All philosophers will attempt to acquire the forks on the left and right hand sides simultaneously during the first step.
LTL	G $(step = 1) \rightarrow (|\langle F_1 \rangle|_{rP_1} = 1 \wedge |\langle F_1 \rangle|_{rP_2} = 1 \wedge |\langle F_2 \rangle|_{rP_2} = 1 \wedge |\langle F_2 \rangle|_{rP_3} = 1 \wedge |\langle F_3 \rangle|_{rP_3} = 1 \wedge |\langle F_3 \rangle|_{rP_4} = 1 \wedge |\langle F_4 \rangle|_{rP_4} = 1 \wedge |\langle F_4 \rangle|_{rP_5} = 1 \wedge |\langle F_5 \rangle|_{rP_1} = 1)$
SPIN LTL	`[] ((step == 1) -> (C[5].x[_rP1] == 1 && C[5].x[_rP2] == 1 && C[6].x[_rP2] == 1 && C[6].x[_rP3] == 1 && C[7].x[_rP3] == 1 && C[7].x[_rP4] == 1 && C[8].x[_rP4] == 1 && C[8].x[_rP5] == 1 && C[9].x[_rP5] == 1 && C[9].x[_rP1] == 1))`
Description	The property guarantees the concurrent behaviour of the model, proving that during the first step all $\langle F_i \rangle$ compartments, $1 \leq i \leq N$ will receive rP_i requests (the multiplicities of rP_i will be one) from their adjacent $\langle Phil_i \rangle$ membranes, each symbolising a philosopher.

Result	Evaluation	Time	Memory	States	State vector
	True	1.25s	157.950 MB	19639	1496 bytes

Table 9.3: Property $P_{phil}1$

The first nine properties documented in this section ($P_{phil}1$ - $P_{phil}9$) have been verified for the instance described in section 9.2, that is $N = 5$ philosophers and forks with each philosopher having associated precisely 3 spaghetti units and 3

Property	A fork can never be shared by any two philosophers. (Verification for adjacent philosophers 1 and 2, having fork f_1 in-between. The generalisation is attained by repeated assertion for instances i, $1 \leq i \leq N$).
LTL	$G \neg(\|\langle Phil_1\rangle\|_{f_1} = 1 \wedge \|\langle Phil_2\rangle\|_{f_1} = 1)$
SPIN LTL	`[] (!(C[0].x[_f1] == 1 && C[1].x[_f1] == 1))`
Description	The multiplicity of object f_1 cannot be equal to one in both compartments $\langle Phil_1\rangle$ and $\langle Phil_1\rangle$ at the same time.

Result	Evaluation	Time	Memory	States	State vector
	True	1.22s	157.950 MB	19639	1496 bytes

Table 9.4: Property $P_{phil}2$

patience objects. It is of course the scenario where a deadlock can be encountered, an affirmation sustained by property $P_{phil}9$ (Table 9.11). These nine formulae confirm the correct behaviour of the model on the one hand, but also identify the problem inexorably associated with the dynamics of distributed system, that is the deadlock state.

The following three properties (Tables 9.12, 9.13 and 9.14) are asserted on a similar instance of the problem, however philosopher 1 is initialised with (and regenerates) a patience value of 4 instead of 3. In this respect, it is proved that a deadlock cannot occur since all philosophers will eventually exhaust their spaghetti. Moreover, a minimum of 19 steps are required for a successful computation of $e\Pi_{phil}$, whose halting configuration is precisely the state where $|\langle Phil_i\rangle|_s = 0$ for $1 \leq i \leq N$. The following theorem is inferred and effectively demonstrated by formal verification:

Theorem 9.1. *A single distinction in the philosophers' behavioural pattern is sufficient to discard the possibility of a deadlock occurrence.*

The final properties, $P_{phil}13$ and $P_{phil}14$ pertain to a configuration of the model where two philosophers ($Phil_1$ and $Phil_3$) are associated a patience value of 4. It has been discovered that the minimum number of steps required to reach the final (accepted) state is reduced to 17 (from 19). Moreover, a successful computation is eventually attained after no more than 45 steps.

There are many ramifications to the study presented in this chapter. On the one hand, it would be interesting to conduct a series of experiments to identify the optimal personality configuration (the *harmony* configuration) such that a

Property	Fork f_1 is only available to philosophers $\langle Phil_1 \rangle$ and $\langle Phil_2 \rangle$.
LTL	G $(|\langle Phil_2 \rangle|_{f_1} = 0 \wedge |\langle Phil_3 \rangle|_{f_1} = 0 \wedge |\langle Phil_4 \rangle|_{f_1} = 0)$
SPIN LTL	`[] (C[2].x[_f1] == 0 && C[3].x[_f1] == 0 && C[4].x[_f1] == 0)`
Description	The property equates to a permanent multiplicity of 0 for object f_1 in all compartments other than $\langle Phil_1 \rangle$ and $\langle Phil_2 \rangle$. Extrapolating the property to all philosopher - fork triples demonstrates the arrangement of forks amidst philosophers illustrated in Figure 9.1 and, in conjunction with the preceding claim (Table 9.4), underpins the consistency of our model.

	Evaluation	Time	Memory	States	State vector
Result	True	1.24s	157.950 MB	19639	1496 bytes

Table 9.5: Property $P_{phil}3$

minimum number of steps is required for the five philosophers to deplete the spaghetti in their bowls. What would the worse-case scenario (the maximum number of steps required to eventually reach the acceptance state) be in this case?

On the other hand, the model extensions described in the preceding section are also valid paths of investigation. A performance evaluation of the model checking technique is particularly valuable when considering supplementary non-deterministic dynamics in the algorithm, such as the arbitrary thinking time or variable patience replenishment.

For the purpose of this thesis, we have demonstrated a modelling approach based on elementary P systems for a representative *synchronisation problem*, but also the feasibility and pertinence of SPIN model checking for concurrent systems specified using an EPS formal model.

Property	If a philosopher with outstanding spaghetti has acquired a single fork and has run out of patience, then he will immediately return this fork and 'replenish' his patience (Verification is conducted for philosopher $Phil_1$ and fork f_5; it is generalised by model checking the same property for all philosophers and all forks individually.)
LTL	$G\ (\lvert\langle Phil_1\rangle\rvert_p = 0 \wedge \lvert\langle Phil_1\rangle\rvert_s > 0 \wedge \lvert\langle Phil_1\rangle\rvert_{f_5} > 0 \wedge \lvert\langle Phil_1\rangle\rvert_{f_1} = 0 \rightarrow X\ (\lvert\langle Phil_1\rangle\rvert_{f_5} = 0 \wedge \lvert\langle Phil_1\rangle\rvert_p > 0))$
SPIN LTL	`[] ((C[0].x[_p] == 0 && C[0].x[_s] > 0 && C[0].x[_f5] > 0 && C[0].x[_f1] == 0) -> X (C[0].x[_f5] == 0 && C[0].x[_p] > 0))`
Description	The property validates a crucial aspect of this algorithm, namely that each philosopher will eventually return a fork if he was unsuccessful in acquiring both after a set amount of time, or more plainly, for as long as he is patient. Thus, when the number of p objects is 0, the philosopher cannot wait any longer and *by the next state* (operator X), the acquired fork is no longer in his possession. It is important to specify the singular fork condition in the left hand side term of the implication, that is, the other expected fork is not present in the compartment which identifies philosopher $Phil_1$: $\lvert\langle Phil_1\rangle\rvert_{f_1} = 0$.

Result	Evaluation	Time	Memory	States	State vector
	True	1.37s	158.837 MB	20290	1496 bytes

Table 9.6: Property $P_{phil}4$

Property	Having acquired both forks, a philosopher will immediately eat one unit of its remaining spaghetti. (The property is documented for philosopher $Phil_2$ and the case when there are three units of spaghetti remaining. Its generalisation is derived when asserted for all $Phil_i, 1 \leq i \leq N$) and $1 \leq j \leq 3$ spaghetti.)					
LTL	G $((|\langle Phil_2\rangle|_{f_1} = 1 \wedge |\langle Phil_2\rangle|_{f_2} = 1 \wedge |\langle Phil_2\rangle|_s = 3) \rightarrow$ X $(|\langle Phil_2\rangle|_s = 2))$					
SPIN LTL	`[] ((C[1].x[_f1] == 1 && C[1].x[_f2] == 1 && C[1].x[_s] == 3) -> X (C[1].x[_s] == 2))`					
Result		Evaluation	Time	Memory	States	State vector
		True	1.18s	158.542 MB	20015	1496 bytes

Table 9.7: Property $P_{phil}5$

Property	If an available fork is solicited, then it is immediately delivered (object is sent) to one of the adjacent philosophers. (Property verified for F_3)					
LTL	G $(((|\langle F_3\rangle|_{rP_3} > 0 \vee |\langle F_3\rangle|_{rP_4} > 0) \wedge |\langle F_3\rangle|_f = 1) \rightarrow$ X $(|\langle F_3\rangle|_f = 0))$					
SPIN LTL	`[] (((C[7].x[_rP3] > 0 || C[7].x[_rP4] > 0) && C[7].x[_f] == 1) -> X (C[7].x[_f] == 0))`					
Result		Evaluation	Time	Memory	States	State vector
		True	1.27s	165.046 MB	24537	1496 bytes

Table 9.8: Property $P_{phil}6$

Property	A philosopher will never acquire its adjacent forks once all spaghetti units have been depleted. (Verification for philosopher $Phil_1$)
LTL	G $(\vert\langle Phil_1\rangle\vert_s = 0 \rightarrow (\vert\langle Phil_1\rangle\vert_{f_1} = 0 \wedge \vert\langle Phil_1\rangle\vert_{f_5} = 0)$
SPIN LTL	`[] (C[0].x[_s] == 0 -> (C[0].x[_f1] == 0 && C[0].x[_f5] == 0))`

Result	Evaluation	Time	Memory	States	State vector
	True	1.17s	157.950 MB	19639	1496 bytes

Table 9.9: Property $P_{phil}7$

Property	A philosopher cannot eat spaghetti unless both forks have been acquired. (Verification for $Phil_2$).
LTL	G $(\vert\langle Phil_2\rangle\vert_s = 3 \wedge (\vert\langle Phil_2\rangle\vert_{f_1} = 0 \vee \vert\langle Phil_2\rangle\vert_{f_2} = 0)) \rightarrow$ X $(\vert\langle Phil_2\rangle\vert_s = 3)$
SPIN LTL	`[] ((C[1].x[_s] == 3 && (C[1].x[_f1] == 0 || C[1].x[_f2] == 0)) -> X (C[1].x[_s] == 3))`
Description	Both left hand side and right hand side fork objects (f_1 and f_2 in this case) must be present in the philosopher's compartment in order to consume s (spaghetti) objects. This is illustrated in the above property by the constant multiplicity three of object s inside $\langle Phil_2\rangle$ between two consecutive states, when either f_1 or f_2 is missing.

Result	Evaluation	Time	Memory	States	State vector
	True	1.37s	160.907 MB	21660	1496 bytes

Table 9.10: Property $P_{phil}8$

Property	There exists a possibility of starvation, such that philosophers can never finish their spaghetti.
LTL	F $(\mathit{step} > -1 \wedge \lvert\langle \mathit{Phil}_1\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_2\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_3\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_4\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_5\rangle\rvert_s = 0$
SPIN LTL	`<> (step > -1 && C[0].x[_s] == 0 && C[1].x[_s] == 0 && C[2].x[_s] == 0 && C[3].x[_s] == 0 && C[4].x[_s] == 0)`
Description	Since a deadlock in the strict sense represents a halting condition for P systems, starvation is signified by an infinite recurrence of a system state (or set of states) which is not *final* or *accepted* for the modelled problem. The property specified in this table identifies starvation by consequence, that is, the unattainable state where all philosophers have depleted the units of spaghetti. Since there is no existential quantifier in LTL to express the formula directly (i.e. there exists a path where it is never the case the philosophers deplete their spaghetti), we can infer this assertion by the 'False' evaluation of an *eventually* statement. More precisely, the negative implication can be translated as: It is not the case that all philosophers will eventually consume their spaghetti entirely. The $\mathit{step} > -1$ term is necessarily appended to the conjunction in the *finally* claim, to indicate that only states which succeed the initialisation stage are pertinent to the verification process. At step -1 all arrays inside type definitions are initialised with a value of zero which would render the property incorrectly valid. This state is always extraneous to the search conducted by SPIN and its exclusion is explicit in the LTL formula.

Result	Evaluation	Time	Memory	States	State vector
	False	0.01s	128.830 MB	6	1496 bytes

Table 9.11: Property $P_{\mathit{phil}}9$

Property	All philosophers will eventually deplete (consume) their spaghetti units, if at least one philosopher is more (or less) patient than the others.
LTL	F $(step > -1 \wedge \lvert\langle Phil_1\rangle\rvert_s = 0 \wedge \lvert\langle Phil_2\rangle\rvert_s = 0 \wedge \lvert\langle Phil_3\rangle\rvert_s = 0 \wedge \lvert\langle Phil_4\rangle\rvert_s = 0 \wedge \lvert\langle Phil_5\rangle\rvert_s = 0$
SPIN LTL	`<> (step > -1 && C[0].x[_s] == 0 && C[1].x[_s] == 0 && C[2].x[_s] == 0 && C[3].x[_s] == 0 && C[4].x[_s] == 0)`
Description	The very same formula validates the algorithm, confirming the reachability of the acceptance state, which is the moment when all philosophers have zero s objects remaining in their representative compartment. The occurrence of a deadlock is therefore disproved when philosopher $Phil_1$ starts with (and replenishes) a patience value of 4, rather than 3. The overall conclusion drawn is that *one distinction in the behavioural pattern (or personality) of an entity is sufficient to avoid a deadlock* in the proposed model.

Result	Evaluation	Time	Memory	States	State vector
	True	5.05s	182.488 MB	36541	1496 bytes

Table 9.12: Property $P_{phil}10$

Property	It takes a minimum of 19 steps to reach a halting configuration, that is the state where all philosophers have consumed their spaghetti.
LTL	G $((step > -1 \land step < 19) \rightarrow (\lvert\langle Phil_1\rangle\rvert_s = 0 \land \lvert\langle Phil_2\rangle\rvert_s = 0 \land \lvert\langle Phil_3\rangle\rvert_s = 0 \land \lvert\langle Phil_4\rangle\rvert_s = 0 \land \lvert\langle Phil_5\rangle\rvert_s = 0))$
SPIN LTL	`[] ((step > -1 && step < 19) -> !(C[0].x[_s] == 0 && C[1].x[_s] == 0 && C[2].x[_s] == 0 && C[3].x[_s] == 0 && C[4].x[_s] == 0))`
Description	We recall that it is impossible to formulate a claim which identifies a single path in the search tree using linear time temporal logic, directly. It is with CTL's existential quantifier that such a property can be formally expressed. We can devise, however, the property's complement and prove its fallacy. Thus, it is sufficient to demonstrate *it is not* always the case that all states between 0 and 19 do not satisfy the halting condition, that is all philosophers must have depleted their spaghetti units. The counter-example which SPIN finds to disprove our claim is precisely the path the this accepted state. The formula in this table is the expression of the negated property we wish to verify. A secondary affirmation is required to support the claim for *minimal* requirement, and this is illustrated in the following depiction 9.14. The property entails that for the first 18 computational steps, it is indeed the case that philosophers cannot all exhaust their spaghetti.

Result	Evaluation	Time	Memory	States	State vector
	False	1.68s	174.210 MB	30719	1496 bytes

Table 9.13: Property $P_{phil}11$

Property	The philosophers cannot all exhaust their spaghetti during the first 18 computational steps.
LTL	G $((\mathit{step} > -1 \wedge \mathit{step} < 18) \rightarrow (\lvert\langle \mathit{Phil}_1\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_2\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_3\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_4\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_5\rangle\rvert_s = 0))$
SPIN LTL	`[] ((step > -1 && step < 18) -> !(C[0].x[_s] == 0 && C[1].x[_s] == 0 && C[2].x[_s] == 0 && C[3].x[_s] == 0 && C[4].x[_s] == 0))`

Result	Evaluation	Time	Memory	States	State vector
	True	3.15s	207.470 MB	53967	1496 bytes

Table 9.14: Property $P_{phil}12$

Property	It takes a minimum 17 computational steps to reach an acceptance state, when two philosophers have increased patience values (of 4).
LTL	G $((\mathit{step} > -1 \wedge \mathit{step} < 18) \rightarrow (\lvert\langle \mathit{Phil}_1\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_2\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_3\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_4\rangle\rvert_s = 0 \wedge \lvert\langle \mathit{Phil}_5\rangle\rvert_s = 0))$
SPIN LTL	`[] ((step > -1 && step < 18) -> !(C[0].x[_s] == 0 && C[1].x[_s] == 0 && C[2].x[_s] == 0 && C[3].x[_s] == 0 && C[4].x[_s] == 0))`
Description	The same deductive pattern is applied for a different instance of the problem with $\mathit{patience}(1) = 4$ and $\mathit{patience}(3) = 4$ as opposed to 3. The validity of this experiment suggests a certain variation in the 'personality' of philosophers may lead to improved 'efficiency' in resource sharing.

Result	Evaluation	Time	Memory	States	State vector
	False	0.42s	142.429 MB	8958	1496 bytes

Table 9.15: Property $P_{phil}13$

Property	A maximum of 45 steps is required to reach an acceptance state which is also a halting configuration.
LTL	G $((step < 45) \vee (\lvert\langle Phil_1\rangle\rvert_s = 0 \wedge \lvert\langle Phil_2\rangle\rvert_s = 0 \wedge \lvert\langle Phil_3\rangle\rvert_s = 0 \wedge \lvert\langle Phil_4\rangle\rvert_s = 0 \wedge \lvert\langle Phil_5\rangle\rvert_s = 0))$
SPIN LTL	`[] (step < 45 || (C[0].x[_s] == 0 && C[1].x[_s] == 0 && C[2].x[_s] == 0 && C[3].x[_s] == 0 && C[4].x[_s] == 0))`
Description	The step variable is always incremented whenever at least one rule has been in applied in $e\Pi_{phil}$.

	Evaluation	Time	Memory	States	State vector
Result	True	0.53s	145.238 MB	10928	1496 bytes

Table 9.16: Property $P_{phil}14$

Chapter 10

Quantitative analysis of non-probabilistic models: A biased coin toss example

10.1 Objectives

In this chapter we present a formal verification approach suitable for stochastic models, whose behaviour is generally investigated by means of probabilistic model checking. The software tools which facilitate such examinations in an automated way, must be capable of computing probability updates for each state transition. As such, the model checker not only generates a system's complete state space, but also performs the probability calculus relevant to the random process the model implements.

A probabilistic model checker asserts the validity of a property with direct reference to the probability of the underlying state (or states) being reached. Properties such as: *An event x will eventually occur with probability p* or *What is the probability that process p1 terminates before process p2?* not only express a behavioural trait of the model, but also convey *quantitative* information about its weight or likelihood.

A distinct goal of this project was to investigate the viability and relevance of our formal verification approach for probabilistic models. Whilst elementary P systems cannot directly implement such models, it turns out that a basic quantitative analysis can nevertheless be formally conducted on EPS instances, using LTL model checking with SPIN. A biased coin toss example underpins the methodology described in this chapter and is sufficient to emphasize the inherent limitations of this strategy.

10.2 EPS Model

We consider the following EPS model implementing a biased coin toss experiment, where the probability of a 'heads' result is 0.7 and complementary, a 'tails' outcome will occur with 0.3 probability.

$$\Pi_{coin} = \{O, C, R\}$$

where

$O = \{c, h, t, p, b\};$

$C = \{(c, p)\};$

$R = \{$
$\quad c \longrightarrow h, c;$
$\quad c \longrightarrow t, c;$
$\quad \langle h \rangle : p \longrightarrow 7p;$
$\quad \langle t \rangle : p \longrightarrow 3t;$
$\quad t \longrightarrow T, b;$
$\quad h \longrightarrow H, b;$
$\}$

The following listing accompanies the formal definition of $e\Pi_{coin}$ with its representation in the eps notation, as required by the *eps-tools* parser.

```
(c, p);

c -> h, c;
c -> t, c;

h: p -> 7p;
t: p -> 3p;

t -> T, b;
h -> H, b;
```

The model consists of a single compartment with two main procedures:

1. A non-deterministic choice between the two outcomes, *heads* and *tails*, of the coin toss and

2. Probability update rules which are applied in accordance to the event that has occurred.

The first two rules, $c \longrightarrow h, c$ and $c \longrightarrow t, c$, symbolise a coin toss which non-deterministically results in either heads or tails. The object c represents the coin and is utilised as a catalyst in both rules, whilst objects h and t denote the probabilistic events *heads* and *tails* of the coin toss experiment, respectively.

We underline the fact that, although two probabilistic events are being modelled in $e\Pi_{coin}$, the model itself is *not* probabilistic as it does not provide any means for specifying the likelihood of each event (i.e. each rule generating a h or t object). We recall that stochastic P systems augment multiset rewriting and communication with *kinetic stochastic constants* which, in conjunction with the multiplicities of objects on the left hand side, can be used to dynamically compute the probability associated with each rule. By contrast, elementary P systems do not include such provision and the occurrence of probabilistic events is reduced to non-deterministic selection in our model. As we will see in the following section, this is indeed sufficient to perform a basic quantitative analysis for this type of random process, by LTL model checking.

The next two rules $\langle h \rangle : p \longrightarrow 7p$ and $\langle h \rangle : p \longrightarrow 3t$ represent the *probability updates*[1] which reflect the *sequential occurrence* of any of the two events. Since *heads* and *tails* are independent events, the simplified multiplication rule is used to compute the probabilities for each subsequent coin toss. For instance, if the probability of the *heads* event is $P(heads) = 0.7$ and the probability of *tails* is $P(tails) = 0.3$, then the probability of obtaining the sequence $(heads, tails, heads)$ over three coin tosses is equal to the product $P(heads) \times P(tails) \times P(heads) = 0.147$. Figure 10.1 depicts the complete probability distribution of our coin toss experiment, for three consecutive tosses. Each arrow represents the occurrence of a probabilistic event (*heads* or *tails*) whereas the vertices envelop the probability of this occurrence.

We note that P systems generally operate with objects whose multiplicities are represented by positive integers. Consequently, it is not possible to specify probability values (rational numbers) as multiplicities without resorting to an artifice: the computed probability product is simply divided by 10^b, where b represents the total number of decimal places introduced in the multiplication. For example, if $P_1 = 0.05$ and $P_2 = 0.95$, then each time a multiplication with P_1 is effected, two b objects are generated, whereas, for a P_2 product, one b is added. The EPS rules which support this technique in our model are $t \longrightarrow T, b$ and $h \longrightarrow H, b$. The outcome of these rules, together with the p yield from the previous rewrites, suggest that either a *tails* (T) or a *heads* (H) event has

[1]Importantly, the updates we are referring to do not apply on the initial probabilities $P(heads)$ and $P(tails)$; these are constant values because the two events are independent.

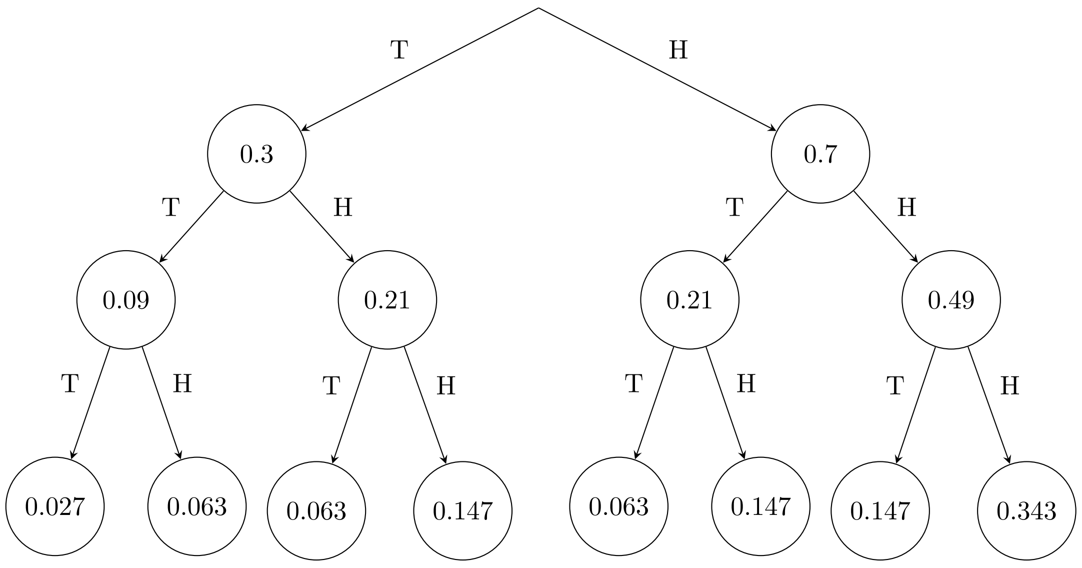

Figure 10.1: Expanded probability distribution for the biased coin toss experiment modelled by $e\Pi_{coin}$

occurred with a probability of $p/10^b$. This is formally captured in the following remark:

Remark 10.1. The EPS model, $e\Pi_{coin}$ computes the probability of an arbitrary sequence of outcomes from $\{heads, tails\}$ in a coin tossing experiment. The sequence is not directly represented or recorded, however, the number of *heads* and *tails* outcomes is encoded by the multiplicities of objects H and T respectively. The probability of a sequence containing x *heads* and y *tails* is denoted by the ratio $|\langle c\rangle|_p/10^{|\langle c\rangle|_b}$ at the computational step where $|\langle c\rangle|_H = x$ and $|\langle c\rangle|_T = y$.

Whilst an $e\Pi_{coin}$ computation pertains to a specific probabilistic event sequence, a model checker must generate a system's complete state space such that any verification (i.e. search operation) undertaken is exhaustive. The non-deterministic choice between the two coin toss events in our model achieves precisely this goal. We recall that non-deterministic branching equates to a combinatorial state expansion in the context of model checking. All computational paths exhibited by $e\Pi_{coin}$ are therefore generated by virtue of the formal verification method itself and not due to explicit productions in the model[1]. Consequently, the model can be subjected to queries pertaining any such path, as if it were interpreted by a probabilistic model-checker, however, it is the query types that

[1]Such an expansion is certainly supported by elementary P system, using membrane instantiation, analogously to the Subset Sum example presented in chapter 8. It is, however, unnecessary if the primary objective of the experiment is formal verification by means of model checking

are inherently limited by the LTL model checker. This matter will be addressed in more detail in the following section.

Step/Scope	$\langle c\rangle$
0	c, p
1	c, p, t
2	T, b, c, p^3, t
3	T^2, b^2, c, p^9, h
4	$H, T^2, b^3, c, p^{63}, h$
5	$H^2, T^2, b^4, c, p^{441}, h$

Table 10.1: Five step simulation of $e\Pi_{coin}$

Finally, we present a listing of the first five computational steps in an *epss* executed simulation (Fig. 10.1). We draw attention on the interpretation of results generated in each configuration: the presence of x objects H and y objects T in the compartment does not signify that the computed probability is associated to *all* scenarios of a certain length (i.e. number of coin tosses) which include x heads and y tails outcomes, but rather this is the probability of a *particular sequence* of $x + y$ elements, with *x heads* and *y tails* occurrences.

10.3 Verification results

The EPS model which implements our coin toss experiment is non-halting and exhibits an unbounded number of states. An adjustment is necessary, in order to satisfy the model checker's finite state space requirement. As exemplified in earlier chapters, the maximum number of steps $e\Pi_{coin}$ is allowed to execute is restricted to ten and as such, an exhaustive search performed by SPIN will be implicitly bound to this constraint.

Another noteworthy amendment of decisive consequence is the substitution of the default procedures generated by the *eps2spin* translation tool for the probability update rules: $\langle h\rangle : p \longrightarrow 7p$ and $\langle t\rangle : p \longrightarrow 3p$. Whilst the repetitive execution of these rules for very high multiplicities of p may itself raise concern and is sufficient to motivate an optimised alternative, an exponential state growth compromises the viability of the method entirely. This occurs due to the non-determinism between one of the aforementioned rules and the creation of b and T or H objects: $t \longrightarrow T, b$ and $h \longrightarrow H, b$. A very simple solution is implemented which avoids the issue and re-establishes the feasibility of model checking $e\Pi_{coin}$. Since we know that the two probability updates are mutually exclusive and each rule must exhaust the collection of p objects, we can substitute the iterative rewriting in Promela's sequential model of $e\Pi_{coin}$:

```
now.C[cIndex].x[_p] -= 1;
now.C[cIndex].y[_p] += 3;
```

with the following code:

```
short p = now.C[cIndex].x[_p];
now.C[cIndex].x[_p] = 0;
now.C[cIndex].y[_p] += 3 * p;
```

This artifice reduces the number of applications of the respective procedure (corresponding to rule $\langle t \rangle : p \longrightarrow 3p$) to one, permanently. The rule will be applicable once only per step since the number of ps becomes zero after its initial application. In simple terms, we have substituted the repetitive summation[1] of a constant value with multiplication, in the Promela model.

The first three properties (Tables 10.2, 10.3 and 10.4) are qualitative assertions which underpin the correctness of the model. The following claims, $P_{coin}4-7$ explicitly address the validity of probabilities (numeric values) for specific sequences of events. Such a sequence is represented by a unique computational path in our model, which is naturally generated by SPIN in response to the non-deterministic choice between the two events (*heads* and *tails*). Each system state produced by the model checker includes a configuration of $e\Pi_{coin}$ which primarily consists of:

- one or more objects from $\{H,\, T\}$, each symbolising the occurrence of a probabilistic event, *heads* or *tails*, and
- a pair of values, whose fractional representation equate to the probability of a sequence of events ending in this state.

Consequently, verifying the probability of a particular sequence entails the search for a single state. It is for this reason that LTL model checking can be utilised to conduct an albeit limited quantitative analysis.

[1]It is important to underline that there is no concept of rule repetition (i.e. iteration) in P systems: rules are applied in a maximally parallel manner, which entails a unitary state transition. Thus, a rule $p \longrightarrow 3p$ does not execute repetitively (sequentially) seven times in a compartment with p^7, rather seven *simultaneous applications* are effected during the same computational step - there are no intermediate transitions.

```
short HTS[MAX_STEPS] = 0;
```

Figure 10.2: Declaration of the *heads/tails sequence* array in the Promela model's state vector.

Since this example features independent events, the probabilities computed for sequences with the same number of 'heads' and 'tails' outcomes are identical. Hence, by specifying one H and one T object in the LTL formula, we can enquire about both (*heads*, *tails*) and (*tails*, *heads*) sequences. If there is a requirement to identify a sequence explicitly, then the state vector is supplemented with an array whose purpose is to progressively store the event occurrences (Fig.10.2). In doing so, each state is provided with complete information about the succession of 'heads' and 'tails' results, up to its occurrence. Property $P_{coin}6$ (Table 10.7) demonstrates how such a state can be referenced in an LTL expression.

Finally, $P_{coin}7$ (Table 10.8) highlights a boundary which non-probabilistic model checking techniques cannot transcend. *It is impossible to express a property which requires a value be computed over 'sibling states' generated by a traditional model checker.* In the context of our coin toss experiment, we cannot contrive a formula for a statement such as: 'The probability of the third toss resulting in heads is 0.7'. The verification of such a property entails a summation of values which span over multiple states (of the same depth). This functionality is generally not available to non-probabilistic verification tools (including SPIN) and consequently, the range of properties which can be verified in adherence to this methodology is inherently limited.

The study presented in this chapter demonstrates that, although elementary P systems are not probabilistic modelling frameworks, they can be used to express random processes rigorously, for the purpose of formal verification by model checking. Whilst the technique cannot aspire to the efficiency of specialised algorithms pertaining probabilistic model checkers, it does present certain advantages, such as the flexibility to implement the probability updates according to any desired strategy, the ability to augment the system states with *derived* or *accumulated* data which can be referenced in an LTL formula, and most notably, the benefit of using the unaltered EPS formalism and associated software tools which facilitate an automatic translation to the Promela specification.

Property	At least one event of type *heads* or *tails* is registered in $e\Pi_{coin}$ starting with step 2.
LTL	G $(step < 2 \vee (\lvert\langle c\rangle\rvert_T > 0 \wedge \lvert\langle c\rangle\rvert_H > 0))$
SPIN LTL	`[] (step < 2 || (C[0].x[_T] > 0 || C[0].x[_H] > 0))`
Description	The occurrence of the two probabilistic events is denoted by the presence of objects T (tails) and H (heads) in the compartment identified by scope $\langle c\rangle$. Although a non-deterministic choice between the two outcomes is performed during each step (including step 1), the probability update can only be computed one step after, since it is determined by the prevalent outcome. The event is always registered in conjunction with the probability of its occurrence, denoted by the multiplicities of objects p and b.

Result	Evaluation	Time	Memory	States	State vector
	True	0.01s	128.827 MB	2048	88 bytes

Table 10.2: Property $P_{coin}1$

Property	For a six step execution of $e\Pi_{coin}$ there are five consecutive coin tosses exercised and hence, a maximum of five 'heads' or 'tails' outcomes can be observed.
LTL	G $(step = 6) \rightarrow (\lvert\langle c\rangle\rvert_T < 6 \wedge \lvert\langle c\rangle\rvert_H < 6)$
SPIN LTL	`[] (step == 6 -> (C[0].x[_T] < 6 && C[0].x[_H] < 6))`

Result	Evaluation	Time	Memory	States	State vector
	True	0.01s	128.827 MB	2048	88 bytes

Table 10.3: Property $P_{coin}2$

Property	From a sequence with one *heads* and one *tails* occurrence, the next outcome immediately recorded (i.e. in the next state) is either *heads* or *tails*, but not both.
LTL	G $(\lvert\langle c\rangle\rvert_T = 1 \wedge \lvert\langle c\rangle\rvert_H = 1) \rightarrow$ X $((\lvert\langle c\rangle\rvert_T = 2 \wedge \lvert\langle c\rangle\rvert_H = 1) \vee (\lvert\langle c\rangle\rvert_T = 1 \wedge \lvert\langle c\rangle\rvert_H = 2))$
SPIN LTL	`[] ((C[0].x[_T] == 1 && C[0].x[_H] == 1) -> X ((C[0].x[_T] == 2 && C[0].x[_H] == 1) || (C[0].x[_H] == 2 && C[0].x[_T] == 1)))`
Description	The assertion of this property guarantees that there can only be one outcome at a certain point in an event sequence, by referencing the multiplicities of objects H and T in the compartment identified by scope $\langle c\rangle$ directly. Moreover, it is implied that a coin toss result is generated each computational step.

Result	Evaluation	Time	Memory	States	State vector
	True	0.01s	128.827 MB	2056	88 bytes

Table 10.4: Property $P_{coin}3$

Property	The probability of the first two tosses resulting in 'tails' is 0.09.
LTL	G $(\lvert\langle c\rangle\rvert_T = 2 \wedge \lvert\langle c\rangle\rvert_H = 0) \rightarrow (\lvert\langle c\rangle\rvert_p = 9 \wedge \lvert\langle c\rangle\rvert_b = 2)$
SPIN LTL	`[] ((C[0].x[_T] == 2 && C[0].x[_H] == 0) -> (C[0].x[_p] == 9 && C[0].x[_b] == 2))`
Description	The unique configuration which identifies this state is underpinned by the presence of two Ts and no H objects. The probability of 0.09 is attested by the multiplicities of objects p and b being equal to 9 and 2 respectively, denoting the fraction $9/10^2$.

Result	Evaluation	Time	Memory	States	State vector
	True	0.01s	128.827 MB	2048	88 bytes

Table 10.5: Property $P_{coin}4$

Property	A sequence consisting of two *heads* and two *tails* events occurs with 0.0441 probability.
LTL	G $(\vert\langle c\rangle\vert_T = 2 \wedge \vert\langle c\rangle\vert_H = 2) \rightarrow (\vert\langle c\rangle\vert_p = 441 \wedge \vert\langle c\rangle\vert_b = 4)$
SPIN LTL	`[] ((C[0].x[_T] == 2 && C[0].x[_H] == 2) -> (C[0].x[_p] == 441 && C[0].x[_b] == 4))`
Description	The property is consistent for this model because the *heads* and *tails* events are independent. This implies that all sequences of four elements having two 'heads' and two 'tails' outcomes occur with the same probability. Hence, it is sufficient to specify the number of events of each type in a configuration to determine the probability for an individual sequence. The value is conveyed by the ratio $441/10^4$.

	Evaluation	Time	Memory	States	State vector
Result	True	0.01s	128.827 MB	2048	88 bytes

Table 10.6: Property $P_{coin}5$

Property	The sequence of events $(heads, tails, heads, heads)$ occurs with 0.1029 probability.
LTL	G $(HTS[0] = H \wedge HTS[1] = T \wedge HTS[2] = H \wedge HTS[3] = H \wedge step = 5) \rightarrow (\vert\langle c\rangle\vert_p = 1029 \wedge \vert\langle c\rangle\vert_b = 4)$
SPIN LTL	`[] ((HTS[0] == _H && HTS[1] == _T && HTS[2] == _H && HTS[3] == _H && step == 5) -> (C[0].x[_b] == 4 && C[0].x[_p] == 1029))`
Description	Since LTL model checkers do not generally provide *state accumulator functions*, such that during traversal, the state vector is not considered in isolation and formulae could also reference values quantified along the search path (for instance, the sum of all variables x on a particular path), it is required that all *progress* made up to a certain point along the path be stored in each individual state. Specifically, we have integrated an array of length equal to the constant `MAX_STEPS` in order to support the precise referencing of individual sequences (declaration is depicted in Fig. 10.2). Each system state will thus provide information about every 'heads' or 'tails' occurrence up to its encounter: a predicate $HTS[1] = T$ asserts whether the second coin toss in the sequence resulted in 'tails'; $HTS[2] = H$ posits that the third outcome in the sequence is 'heads'. As established previously (for property $P_{coin}5$), it is not strictly necessary to refer to the specific sequence to infer its probability *for independent events*. The example is, however, illustrative for cases where the order in which events occur is consequential. We also note, it is important to specify the length of the sequence in the LTL formulae, otherwise all sequences which stem from this configuration will also be considered in the search operation. This is achieved by inclusion of $step = 5$ in the left hand side term of the implication, which requires that only sequences of length four are inspected.

Result	Evaluation	Time	Memory	States	State vector
	True	0.01s	128.827 MB	2048	88 bytes

Table 10.7: Property $P_{coin}6$

Property	The probability of the sixth coin toss resulting in 'heads' is minimum 0.000729 and maximum 0.117649, for a particular sequence.
LTL	G $(HTS[5] = H \wedge step == 7) \rightarrow (\vert\langle c\rangle\vert_b = 6 \wedge \vert\langle c\rangle\vert_p \geq 729 \wedge \vert\langle c\rangle\vert_p \leq 117649)$
SPIN LTL	`[] ((HTS[5] == _H && step == 7) -> (C[0].x[_b] == 6 && C[0].x[_p] >= 729 && C[0].x[_p] <= 117649))`
Description	The property uncovers an essential limitation of LTL model checking: it is impossible to express a property which requires a value be computed from multiple states. Thus, it was strictly required to mention that it is a single sequence whose minimum and maximum probabilities of the *heads* event at the sixth toss are $729/10^6$ and $117649/10^6$ respectively. Moreover, due to the same limitation we can only stipulate inequalities and not direct equivalence; for instance it is impossible to verify a property stated as: *The probability of the sixth coin toss resulting in 'heads' is* 0.7 or *Given that the fifth coin toss resulted in 'tails', the probability of the next outcome being 'heads' is* 0.21. We note the increased size of the state vector and total memory required by the model checker for this particular property. Since the numbers generated exceed the size of the *short* data type, all variables were elevated to the *int* type, to accommodate the high values generated (multiplicities of p).

	Evaluation	Time	Memory	States	State vector
Result	True	0.01s	129.022 MB	2048	144 bytes

Table 10.8: Property $P_{coin}7$

Chapter 11

Conclusions

In this thesis we have demonstrated the feasibility of model checking membrane systems. A fundamental distinction between the conventional algebraic representation of parallelism, promoted by process calculi, and the maximally parallel multiset transitions inherent to P systems was identified. Whilst a process algebra, such as CSP, reconciles an agnostic view on parallel dynamics with sequential computation *by reduction* to non-deterministic branching, P systems feature an *expansive* 'maximal parallelism' which integrates all independently executable instructions into a monadic state transition. On this basis, it is shown that process models, which are ubiquitous to the formal verification of concurrent systems, are decidedly inadequate for the representation of membrane systems. Since model checking addresses the state space of a model and not the procedures which generate this state space (i.e. its systematic construction), it stands as a particularly suitable verification technique for P system models. In this respect, an efficient modelling approach for the SPIN model checker was introduced, whereby compartments are not associated a distinct process, nor are these selected non-deterministically for exhaustive execution (of applicable rules). This strategy implicitly circumvents the exponential state expansion due to instruction interleaving.

In chapter 5 we introduced elementary P systems, a distributed model which subsumes the membrane computing paradigm and its vast functional vocabulary. The minimal set of primitives integrated and their operational capacity are sufficiently expressive to support the most prevalent P system variants on the one hand, but also translate to a compact and intuitive Promela specification, required by SPIN, on the other hand.

The formal verification approach proposed in this thesis is supported by several software tools, bridging the theoretical domain of P systems with the practical method of automated model checking. The eps modelling language aims to provide an unambiguous and concise EPS representation, using a very simple syntax. Our *eps-tools* module features a parser for the eps notation, a simulator for elementary P systems and most importantly, a translation tool which targets the Promela specification and implements the modelling principles postulated in chapter 4.

The methodology with the accompanying software tools was progressively demonstrated over five heterogeneous case studies. The two principal objectives of these undertakings were:

1. To emphasize the modelling potential of elementary P systems in the context of membrane computing, identify semantic correspondences relative to their kindred models and highlight noteworthy translation patterns;

2. To corroborate the suitability of model checking P systems in an extended context, with a sharp focus on distributed and parallel computation. The selection includes a structured model, a linear time parallel algorithm for an NP-complete problem and an EPS implementation of a synchronisation problem.

In chapter 6 we exemplified EPS modelling and verification on a simple algorithm which generates numbers from the Fibonacci series. It was noted that reachability properties can be verified for systems with infinite states, since only a subset of their state space is required to prove their faithfulness. This holds if the expected evaluation is 'true', that is, the model satisfies this property. Additionally, we underlined the consequence of permitting instruction interleaving (within compartments only). This was reflected in the number of superfluous states generated by SPIN in the global reachability graph.

Next, we have shown in chapter 7 that elementary P systems are as efficient at modelling structures of compartments as their kindred variants, in the absence of an ancillary provision in the model definition. Rather than including a hierarchical membrane structure primitive (usually denoted by μ) or a set of communication channels resembling edges in a graph-like structure, object pairs in select EPS compartments are used to encode directional and dynamic links between them. Moreover, this encoding scheme also confers a certain flexibility which allows a single object (symbol) to denote a reflexive relation between two compartments (such as parent-child), effectively reducing the size of the state vector in the translated Promela model and thus mitigating the memory requirement for model checking.

In chapter 8 we established that a computationally hard problem which requires exponential resources (relative to the size of the problem, assuming P $\neq$ NP) does not entail an exponential number of states and is consequently no less tractable to verify by means of model checking. An EPS based algorithm for solving the Subset Sum problem in linear time was presented and formally examined. An important quality of elementary P systems was emphasized, namely, the resolute adherence of transition rules to the maximally parallel execution strategy. Membrane instantiation is not achieved by rules applied on compartments (similarly to membrane division rules), but rather a quiescent compartment is assumed during each computational step and activated when it becomes the recipient of a multiset of objects. The principle simplifies the semantics of EPS, which accordingly translates to a coherent, more compact Promela model that can be manipulated effortlessly.

In the following chapter (chapter 9) we presented an innovative implementation of the Dining Philosophers scenario, using elementary P systems. Our solution was formulated on the basis that a sterile plurality of entities has no observable dynamics, and if a projection into a mathematically abstract formal domain is considered, then a synthetic distinction can also be introduced from an *external* context. Hence, an individualised configuration, loosely referred to as 'personality', can be used to determine the behaviour of each philosopher *independently.* The personality pattern may be inferred by a singular value, such as the 'patience' in acquiring both forks, or may encompass other factors as well, for instance a so called thinking time. The fact that the personality complex is pre-computed and no central authority is required to arbitrate the distribution of forks is of key significance. We have formally demonstrated by model checking that a deadlock can indeed occur when all philosophers share the same personality configuration and act identically, indefinitely. Conversely, if at least one distinction exists in the philosophers' behavioural pattern (the 'patience' value of a single philosopher was increased from 3 to 4), then the possibility of a deadlock occurrence is dismissed. Importantly, our model assumed a finite supply of spaghetti; details on how this implementation can be extended to accommodate infinite spaghetti units were also provided in section 9.3. Furthermore, we have also described optimisation strategies applicable directly to the translated Promela model. A very instructive and consequential practice was to discharge the rules creating objects that are not pertinent to the verification process, especially if these compete non-deterministically for execution.

Finally it was shown (in chapter 10) that although elementary P systems are not probabilistic models, it is practically viable to conduct a formal quantitative analysis on systems which implement a simple random process. A biased coin toss experiment, featuring the two independent events, *heads* and *tails*, substantiates our remark. A significant aspect of the study is the hybrid modelling approach for capturing probability values for all possible sequential occurrences, that is, all combinations of the two probabilistic events. On one hand, each 'heads' or 'tails' yield initiates a probability update rule in the model - the likelihood of the occurrence is computed by the model itself; and on the other hand, all computational paths are generated by the model checker tool (SPIN) as a result of the non-deterministic choice between the two independent events in our model. Equally important is the inherent limitation of LTL model checking which was pronouncedly revealed in this example: it is impossible to express a property which requires a value be computed from multiple states situated on disjoint computational paths. This finding circumscribes the suitability of our formal verification approach for non-probabilistic models.

The methodology proposed and demonstrated in this thesis opens many doors to further paths of research and development. On the one hand, formal verification can accompany any future examination on novel P system algorithms and modelling techniques. The sequential modelling principles submitted in this thesis constitute the theoretical foundation for any such undertaking, whilst the elementary P system computational model, the eps modelling language and the *eps-tools* software framework support the approach in a practical manner.

On the other hand, the set of tools which mediates P system formal models and the model checking technique could be enhanced and supplemented. One may consider the possibility of including LTL properties which relate to the elementary P system model and not directly to the encoded Promela representation. Rather than addressing the multiplicities of objects in compartments using the compartment data type and array - `C[0].x[_a]` - the eps language syntax could be extended to allow for *scope references*, translatable to atomic propositions in LTL and possibly supported by ancillary procedures: $|x, !y|(a)$ could reference the number of objects a across all compartments with at least one x and no ys. Having the LTL formulae supplied together with the EPS model as input to the translation tool promotes other (automated) optimisations, in particular the exclusion of objects irrelevant to the verification of the respective properties.

Another fruitful undertaking worth considering is the projection to other model checker tools. One noteworthy contender is nuXsmv [5, 25], a relatively new (released in 2014) symbolic model checker for the analysis of synchronous finite-state and infinite-state systems.

Lastly, an in-depth performance analysis of the formal verification approach set forth in this thesis is of exceptional value. Although the limitations of model checking can generally be asserted relative to the number of processes and interleaved atomic instructions, an assessment in the context of membrane computing, specifically in terms of elementary P system compartments and rules, is instrumental in determining the feasibility of the technique for various sizeable models. Complementary, an investigation on its scalability for typical scenarios (such as exponential compartment instantiation - the Subset Sum problem, non-deterministic distribution of resources - the Dining Philosophers problem) is also of interest. This could be achieved by directly considering the case studies presented in this thesis in various configurations. For example, a Subset Sum problem instance with a set A of 25, 50, 100 etc elements and multiple solutions; a Dining Philosophers instance featuring 7, 9 ... philosophers and a varying supply of spaghetti.

Appendix A

A ten step simulation of an EPS model which generates the numbers in the Fibonacci series

```
Simulation started at step: 0
Step 1:
1. a, 2x
2. b, x
3. c, x
--------------------------------------
Step 2:
1. a, 3x
2. b, 2x
3. c, x
--------------------------------------
Step 3:
1. a, 5x
2. b, 3x
3. c, 2x
--------------------------------------
Step 4:
1. a, 8x
2. b, 5x
3. c, 3x
--------------------------------------
Step 5:
1. a, 13x
2. b, 8x
```

```
3. c, 5x
--------------------------------------
Step 6:
1. a, 21x
2. b, 13x
3. c, 8x
--------------------------------------
Step 7:
1. a, 34x
2. b, 21x
3. c, 13x
--------------------------------------
Step 8:
1. a, 55x
2. b, 34x
3. c, 21x
--------------------------------------
Step 9:
1. a, 89x
2. b, 55x
3. c, 34x
--------------------------------------
Step 10:
1. a, 144x
2. b, 89x
3. c, 55x
--------------------------------------
System halted. Maximum number of steps (10) reached.
Simulation ended at step: 10
```

Appendix B

Promela code for the Fibonacci EPS model generated by the eps2spin translation tool

```
/*
 * EPS: Fibonacci_distributed.eps
 * Generated: 3/7/2016, 9:46:57 PM
 * Converter version: 0.01
 */

#define A_SIZE 4
#define _a 0
#define _x 1
#define _b 2
#define _c 3

#define MAX_COMPARTMENTS 3
#define RULE_COUNT 2

typedef Compartment {
     short x[A_SIZE] = 0;
     short y[A_SIZE] = 0;
     bit rulesApplicable[RULE_COUNT] = 0;
}
Compartment C[MAX_COMPARTMENTS];
short C_COUNT = 3;
c_code {
     #include<stdbool.h>

```

```
    // b
    bool isInScope0(short cIndex) {
        short *x = now.C[cIndex].x;
        return (x[_b] >= 1);
    }
    // c
    bool isInScope1(short cIndex) {
        short *x = now.C[cIndex].x;
        return (x[_c] >= 1);
    }
    // b
    bool isInScope2(short cIndex) {
        short *x = now.C[cIndex].x;
        return (x[_b] >= 1);
    }
    bool atLeastOneInScope2(short excludeSelfIndex) {
        for(short i = 0; i < now.C_COUNT; ++i) {
            if(i != excludeSelfIndex && isInScope2(i)) {
                return true;
            }
        }
        return false;
    }

    bool isRule0ApplicableTo(short cIndex) {
        return isInScope0(cIndex);
    }
    void applyRule0(short cIndex) {
        short x = now.C[cIndex].x[_x];
        now.C[cIndex].x[_x] = 0;
        now.C[cIndex].y[_x] += x;
        for(short i = 0; i < now.C_COUNT; ++i) {
            if(i != cIndex) {
                now.C[i].y[_x] += x;
            }
        }
    }
    bool isRule1ApplicableTo(short cIndex) {
        return isInScope1(cIndex)
         && atLeastOneInScope2(cIndex);
    }
    void applyRule1(short cIndex) {
        short x = now.C[cIndex].x[_x];
        now.C[cIndex].x[_x] = 0;
        for(short i = 0; i < now.C_COUNT; ++i) {
            if(i != cIndex) {
                if(isInScope2(i)) {
                    now.C[i].y[_x] += x;
                }
            }
        }
    }

    void assertRuleApplicability() {
        for(short i = 0; i < now.C_COUNT; ++i) {
            now.C[i].rulesApplicable[0] = isRule0ApplicableTo(i);
            now.C[i].rulesApplicable[1] = isRule1ApplicableTo(i);
        }
    }
    void commit() {
        for(short i = 0; i < now.C_COUNT; ++i) {
            for(short j = 0; j < A_SIZE; ++j) {
                now.C[i].x[j] += now.C[i].y[j]; now.C[i].y[j] = 0;
            }
        }
    }
    bool isLhs0Applicable(short cIndex) {
        short *x = now.C[cIndex].x;
        if(x[_x] < 1) {
            return false;
        }
        return true;
    }
    bool isRule0Applicable(cIndex) {
        return now.C[cIndex].rulesApplicable[0] &&
            isLhs0Applicable(cIndex);
    }
    bool isLhs1Applicable(short cIndex) {
        short *x = now.C[cIndex].x;
        if(x[_x] < 1) {
            return false;
        }
        return true;
    }
    bool isRule1Applicable(cIndex) {
        return now.C[cIndex].rulesApplicable[1] &&
                isLhs1Applicable(cIndex);
     }

}
proctype EPS() {
     short i;
     do
     :: atomic {
          c_code { assertRuleApplicability(); };
          for(i: 0 .. C_COUNT - 1) {
               do
                    :: c_expr { isRule0Applicable(PEPS->i) } ->
                         c_code { applyRule0(PEPS->i); };
                    :: c_expr { isRule1Applicable(PEPS->i) } ->
                         c_code { applyRule1(PEPS->i); };
                    :: else -> break;
               od;
          }
          c_code { commit(); };
     } od;
}
init {
     atomic {
          C[0].x[_a] = 1;
          C[0].x[_x] = 1;
          C[1].x[_b] = 1;
          C[1].x[_x] = 1;
          C[2].x[_c] = 1;
          run EPS();
     }
}
```

Appendix C

Elementary P system model expansion for a DAG node counting algorithm

```
(n1, s, q, a, p_n2, p_n3);
n1: {
    q, a -> [a @ c_n1];
    a: s -> [c @ p_n1];
}

(n2, s, q, p_n4, p_n5, c_n1);
n2: {
    q, a -> [a @ c_n2];
    a: s -> [c @ p_n2];
}

(n3, s, q, p_n5, p_n6, c_n1);
n3: {
    q, a -> [a @ c_n3];
    a: s -> [c @ p_n3];
}

(n4, s, q, c_n2);
n4: {
    q, a -> [a @ c_n4];
    a: s -> [c @ p_n4];
}

(n5, s, q, p_n8, c_n2, c_n3);
```

```
n5: {
    q, a -> [a @ c_n5];
    a: s -> [c @ p_n5];
}

(n6, s, q, p_n7, c_n3, c_n9);
n6: {
    q, a -> [a @ c_n6];
    a: s -> [c @ p_n6];
}

(n7, s, q, p_n8, c_n6);
n7: {
    q, a -> [a @ c_n7];
    a: s -> [c @ p_n7];
}

(n8, s, q, c_n5, c_n7);
n8: {
    q, a -> [a @ c_n8];
    a: s -> [c @ p_n8];
}

(n9, s, q, a, p_n6);
n9: {
    q, a -> [a @ c_n9];
    a: s -> [c @ p_n9];
}
```

Appendix D

Elementary P system representation of the Dining Philosophers problem expressed in the eps modelling language

```
(Phil1, 5p, 3s, q);
(Phil2, 3p, 3s, q);
(Phil3, 3p, 3s, q);
(Phil4, 3p, 3s, q);
(Phil5, 3p, 3s, q);

(F1, f);
(F2, f);
(F3, f);
(F4, f);
(F5, f);

F1: {
    f, rP1 -> [f1 @ Phil1];
    f, rP2 -> [f1 @ Phil2];
    !f: {
        rP1 -> z1;
        rP2 -> z2;
    }
}
F2: {
    f, rP2 -> [f2 @ Phil2];
    f, rP3 -> [f2 @ Phil3];
    !f: {
        rP2 -> z2;
        rP3 -> z3;
    }
}
F3: {
    f, rP3 -> [f3 @ Phil3];
    f, rP4 -> [f3 @ Phil4];
    !f: {
        rP3 -> z3;
        rP4 -> z4;
    }
}
F4: {
    f, rP4 -> [f4 @ Phil4];
    f, rP5 -> [f4 @ Phil5];
    !f: {
        rP4 -> z4;
        rP5 -> z5;
    }
}
F5: {
    f, rP5 -> [f5 @ Phil5];
    f, rP1 -> [f5 @ Phil1];
    !f: {
        rP5 -> z5;
        rP1 -> z1;
    }
}

Phil1: {
    f1, f5, s -> [f @ F1], [f @ F5];
    s: { !f1 | !f5: {
        q, p -> q, [rP1 @ F1], [rP1 @ F5];
        !p: {
            f1 -> [f @ F1];
            f5 -> [f @ F5];
            q -> q, 5p;
        }
    }}
}
Phil2: {
    f2, f1, s -> [f @ F2], [f @ F1];
    s: { !f2 | !f1: {
        q, p -> q, [rP2 @ F2], [rP2 @ F1];
        !p: {
            f2 -> [f @ F2];
            f1 -> [f @ F1];
            q -> q, 3p;
        }
    }}
}
Phil3: {
    f3, f2, s -> [f @ F3], [f @ F2];
    s: { !f3 | !f2: {
        q, p -> q, [rP3 @ F3], [rP3 @ F2];
        !p: {
            f3 -> [f @ F3];
            f2 -> [f @ F2];
            q -> q, 3p;
        }
    }}
}
Phil4: {
    f4, f3, s -> [f @ F4], [f @ F3];
    s: { !f4 | !f3: {
        q, p -> q, [rP4 @ F4], [rP4 @ F3];
        !p: {
            f4 -> [f @ F4];
            f3 -> [f @ F3];
            q -> q, 3p;
        }
    }}
}
Phil5: {
    f5, f4, s -> [f @ F5], [f @ F4];
    s: { !f5 | !f4: {
        q, p -> q, [rP5 @ F5], [rP5 @ F4];
        !p: {
            f5 -> [f @ F5];
            f4 -> [f @ F4];
            q -> q, 3p;
        }
    }}
}
```

References

[1] eps-tools. `https://github.com/ciprian-dragomir/eps`. Accessed: 2016-01-25.

[2] JSON (Javascript Object Notation). `http://json.org`. Accessed: 2016-01-25.

[3] Node.js. `http://nodejs.org`. Accessed: 2016-01-25.

[4] NuSMV: a new symbolic model checker. `http://nusmv.fbk.eu`. Accessed: 2016-04-10.

[5] The nuXmv model checker. `https://nuxmv.fbk.eu`. Accessed: 2016-04-10.

[6] Promela Reference – assert. `http://spinroot.com/spin/Man/assert.html`. Accessed: 2016-01-28.

[7] Standard ecma-262. ecmascript 2015 language specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`. Accessed: 2016-01-25.

[8] Verifying Multi-threaded Software with Spin. `http://spinroot.com`. Accessed: 2016-01-25.

[9] Artiom Alhazov, Rudolf Freund, and Marion Oswald. *Developments in Language Theory: 9th International Conference, DLT 2005, Palermo, Italy, July 4-8, 2005. Proceedings*, chapter Tissue P Systems with Antiport Rules and Small Numbers of Symbols and Cells, pages 100–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[10] Artiom Alhazov, Rudolf Freund, and Marion Oswald. Cell/symbol complexity of tissue p systems with symport/antiport rules. *International Journal of Foundations of Computer Science*, 17(01):3–25, 2006.

[11] Artiom Alhazov, Rudolf Freund, and Yurii Rogozhin. *Membrane Computing: 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers*, chapter Computational Power of Symport/Antiport: History, Advances, and Open Problems, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[12] Artiom Alhazov, Yurii Rogozhin, and Sergey Verlan. Minimal cooperation in symport/antiport tissue p systems. *International Journal of Foundations of Computer Science*, 18(01):163–179, 2007.

[13] Artiom Alhazov and Dragos Sburlan. Static Sorting P Systems. *Applications of Membrane Computing 2006*, pages 215–252, 2006.

[14] Oana Andrei, Gabriel Ciobanu, and Dorel Lucanu. *Membrane Computing: 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, chapter Executable Specifications of P Systems, pages 126–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[15] Oana Andrei, Gabriel Ciobanu, and Dorel Lucanu. A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science*, 373(3):163–181, 2007.

[16] Joshua J. Arulanandham. Implementing Bead-Sort with P systems. In *In Proc. 3rd International Conference on Unconventional Models of Computation, UMC*, pages 115–125. Springer, 2002.

[17] Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo, and Simone Tini. Compositional semantics and behavioral equivalences for p systems. *Theoretical Computer Science*, 395:77–100, 2008.

[18] Jan A. Bergstra and Jan W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77 – 121, 1985.

[19] Francesco Bernardini and Marian Gheorghe. Population P Systems. *Journal of Universal Computer Science*, 10(5):509–539, may 2004.

[20] Francesco Bernardini, Marian Gheorghe, and Natalio Krasnogor. Quorum sensing P systems. *Theoretical Computer Science*, 371(12):20 – 33, 2007.

[21] Daniela Besozzi, Paolo Cazzaniga, Dario Pescini, and Giancarlo Mauri. Modelling metapopulations with stochastic membrane systems. *Biosystems*, 91(3):499 – 514, 2008. P-Systems Applications to Systems Biology.

[22] Julius R. Büchi. *The Collected Works of J. Richard Büchi*, chapter On a Decision Method in Restricted Second Order Arithmetic, pages 425–435. Springer New York, New York, NY, 1990.

[23] Nadia Busi. Using wellstructured transition systems to decide divergence for catalytic p systems. *Theoretical Computer Science*, 372:125–135, 2007.

[24] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.

[25] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *CAV*, pages 334–342, 2014.

[26] Rodica Ceterchi and Carlos Martín-Vide. P Systems with communication for static sorting. *GRLMC Report*, (26), 2003.

[27] Haiming Chen, Rudolf Freund, Mihai Ionescu, Gheorghe Păun, and Mario J. Pérez-Jiménez. On String Languages Generated by Spiking Neural P Systems. *Fundamentae Informaticae*, 75(1-4):141–162, 2007.

[28] Haiming Chen, Mihai Ionescu, Andrei Păun, Gheorghe Păun, and Bianca Popa. On Trace Languages Generated by (Small) Spiking Neural P Systems. In *Pre-Proceedings of 8th Workshop on Descriptional Complexity of formal systems*, June 2006.

[29] Gabriel Ciobanu, Linqiang Pan, Gheorghe Păun, and Mario J. Pérez Jiménez. P systems with minimal parallelism. *Theoretical Computer Science*, 378(1):117–130.

[30] Edmund M. Clarke and Ernst A. Emerson. *25 Years of Model Checking: History, Achievements, Perspectives*, chapter Design and synthesis of synchronization skeletons using branching time temporal logic, pages 196–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[31] Edmund M. Clarke, Ernst A. Emerson, and Aravinda P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. volume 8, pages 244–263, 1986.

[32] Manuel Clavel, Francisco Durán, Steven Eker, Patrick D. Lincoln, Narciso Martí-Oliet, Jose Meseguer, and Jose Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187 – 243, 2002.

[33] Zhe Dang, Oscar H. Ibarra, Cheng Li, and Gaoyan Xie. On the decidability of model-checking for P systems. *Journal of Automata, Languages and Combinatorics*, 11(3):279–298, 2006.

[34] Jürgen Dassow. *Formal Languages and Applications*, chapter Grammars With Regulated Rewriting, pages 249–273. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[35] Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory*. Springer-Verlag, 1989.

[36] Daniel Díaz-Pernil, Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Agustín Riscos-Núñez. Solving Subset Sum in Linear Time by Using Tissue P Systems with Cell Division. In *Proceedings of the 2Nd International Work-conference on The Interplay Between Natural and Artificial Computation, Part I: Bio-inspired Modeling of Cognitive Tasks*, IWINAC '07, pages 170–179, Berlin, Heidelberg, 2007. Springer-Verlag.

[37] Daniel Díaz-Pernil, Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Agustin Riscos-Núñez. A uniform family of tissue P systems with cell division solving 3-COL in a linear time. *Theoretical Computer Science*, 404(1-2):76–87, 2008.

[38] Ciprian Dragomir, Florentin Ipate, Savas Konur, Raluca Lefticaru, and Laurentiu Mierla. *Membrane Computing: 14th International Conference, CMC 2013, Chişinău, Republic of Moldova, August 20-23, 2013, Revised Selected Papers*, chapter Model Checking Kernel P Systems, pages 151–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[39] Ciprian Dragomir, Florentin Ipate, Savas Konur, Raluca Lefticaru, and Laurentiu Mierla. Model Checking Kernel P Systems. In Artiom Alhazov, Svetlana Cojocaru, Marian Gheorghe, Yurii Rogozhin, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 8340 of *Lecture Notes in Computer Science*, pages 151–172. Springer Berlin Heidelberg, 2014.

[40] Ernst A. Emerson and Edmund M. Clarke. *Automata, Languages and Programming: Seventh Colloquium Noordwijkerhout, the Netherlands July 14–18, 1980*, chapter Characterizing correctness properties of parallel programs using fixpoints, pages 169–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980.

[41] Rudolf Freund. *Membrane Computing: 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*,

chapter Asynchronous P Systems and P Systems Working in the Sequential Mode, pages 36–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[42] Rudolf Freund and Marion Oswald. *Membrane Computing: International Workshop, WMC-CdeA 2002 Curtea de Arges, Romania, August 19–23, 2002 Revised Papers*, chapter P Systems with Activated/Prohibited Membrane Channels, pages 261–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[43] Rudolf Freund and Andrei Păun. *Membrane Computing: International Workshop, WMC-CdeA 2002 Curtea de Arges, Romania, August 19–23, 2002 Revised Papers*, chapter Membrane Systems with Symport/Antiport Rules: Universality Results, pages 270–287. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[44] Rudolf Freund, Gheorghe Păun, and Mario J. Pérez-Jiménez. Tissue P Systems with Channel States. *Theoretical Computer Science*, 330(1):101–116, January 2005.

[45] Rudolf Freund and Sergey Verlan. A formal framework for static (tissue) P systems. In *LNCS*, volume 4860, pages 271 – 284, 2007.

[46] Pierluigi Frisco and Hendrik Jan Hoogeboom. *Membrane Computing: International Workshop, WMC-CdeA 2002 Curtea de Arges, Romania, August 19–23, 2002 Revised Papers*, chapter Simulating Counter Automata by P Systems with Symport/Antiport, pages 288–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[47] Zsolt Gazdag and Gábor Kolonits. *Membrane Computing: 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers*, chapter A New Approach for Solving SAT by P Systems with Active Membranes, pages 195–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[48] Marian Gheorghe, Florentin Ipate, and Ciprian Dragomir. A Kernel P system. *Tenth Brainstorming Week on Membrane Computing*, vol. I:153–170, 2012.

[49] Marian Gheorghe, Florentin Ipate, Raluca Lefticaru, Mario J. Pérez-Jiménez, Adrian Turcanu, Luis Valencia-Cabrera, Manuel García-Quismondo, and Laurentiu Mierla. 3-Col problem modelling using simple kernel P systems. *International Journal of Computer Mathematics*, 90(4):816–830, 2013.

[50] Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Agustín Riscos-Núñez. A fast P system for finding a balanced 2-partition. *Soft Computing*, 9(9):673–678, 2005.

[51] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, August 1978.

[52] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):275–295, 1997.

[53] Ellis Horowitz and Sartaj Sahni. Computing Partitions with Applications to the Knapsack Problem. *J. ACM*, 21(2):277–292, April 1974.

[54] Mihai Ionescu, Gheorghe Păn, and Takashi Yokomori. Spiking Neural P Systems. *Fundamentae Informaticae*, 71:279–308, 2006.

[55] Florentin Ipate, Raluca Lefticaru, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, and Cristina Tudose. Formal verification of P systems with active membranes through model checking. In Marian Gheorghe, Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa, and Sergey Verlan, editors, *Int. Conf. on Membrane Computing*, volume 7184 of *Lecture Notes in Computer Science*, pages 215–225. Springer, 2011.

[56] Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Formal verification of P systems using Spin. *International Journal of Foundations of Computer Science*, 22(1):133–142, 2011.

[57] Tseren-Onolt Ishdorj and Alberto Leporati. Uniform solutions to SAT and 3-SAT by spiking neural P systems with pre-computed resources. *Natural Computing*, 7(4):519–534, 2008.

[58] Mario J. Pérez Jiménez and Agustin Riscos-Núñez. A Linear-Time Solution to the Knapsack Problem Using P Systems with Active Membranes. In Carlos Martín-Vide, Giancarlo Mauri, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 250–268. Springer Berlin Heidelberg, 2004.

[59] Raluca Lefticaru, Florentin Ipate, Luis Valencia-Cabrera, Adrian Turcanu, Cristina Tudose, Marian Gheorghe, Mario J. Pérez-Jiménez, Ionut M. Niculescu, and Ciprian Dragomir. Towards an integrated approach for model simulation, property extraction and verification of P systems. *Tenth Brainstorming Week on Membrane Computing*, 1:291–318, 2012.

[60] Alberto Leporati and Miguel A. Gutiérrez-Naranjo. Solving SUBSET SUM by Spiking Neural P Systems with Pre-computed Resources. *Fundamenta Informaticae*, 87(1):61–77, 2008.

[61] Alberto Leporati, Claudio Zandron, Claudio Ferretti, and Giancarlo Mauri. Solving Numerical NP-Complete Problems with Spiking Neural P Systems. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 336–352. Springer Berlin Heidelberg, 2007.

[62] Aristid Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18(3):280–315, 1968.

[63] Carlos Martín-Vide, Gheorghe Păun, Juan Pazos, and Alfonso Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295 – 326, 2003. Machines, Computations and Universality.

[64] Robin Milner. *A Calculus of Communicating Systems.* Springer Berlin Heidelberg, 1980.

[65] Robin Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, 5 1999.

[66] Madhu Mutyam, Vaka Jaya Prakash, and Kamala Krithivasan. Rewriting Tissue P Systems. *Journal of Universal Computer Science*, 10(9):1250–1271, sep 2004.

[67] Linqiang Pan and Artiom Alhazov. Solving HPP and SAT by P Systems with Active Membranes and Separation Rules. *Acta Informatica*, 43(2):131–145, 2006.

[68] Linqiang Pan and Carlos Martín-Vide. Solving multidimensional 0 1 knapsack problem by P systems with input and active membranes. *Journal of Parallel and Distributed Computing*, 65(12):1578 – 1584, 2005.

[69] Gheorghe Păun. *Applications of Membrane Computing*, chapter Introduction to Membrane Computing, pages 1–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[70] Mario J. Pérez-Jiménez and Agustín Riscos-Núñez. Solving the Subset-Sum problem by P systems with active membranes. *New Generation Computing*, 23(4):339–356.

[71] Carl Adam Petri and Wolfgang Reisig. Petri net. `http://www.scholarpedia.org/article/Petri_net`, 2008. Accessed: 2015-06-24.

[72] Gordon Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[73] Gordon Plotkin. Structural operational semantics. *Journal of Logic and Algebraic Programming*, 60:17–139, 2004.

[74] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[75] Andrei Păun and Gheorghe Păun. The power of Communication: P Systems with Symport/Antiport. *New Generation Computers*, 20(3):295–306, 2002.

[76] Andrei Păun, Gheorghe Păun, and Grzegorz Rozenberg. Computing by communication in networks of membranes. *International Journal of Foundations of Computer Science*, 13(06):779–798, 2002.

[77] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[78] Gheorghe Păun. P systems with active membranes: Attacking NP-complete problems. *J. Automata, Languages, Combinatorics*, 6:75–90, 2001.

[79] Jean P. Queille and Joseph Sifakis. *International Symposium on Programming: 5th Colloquium Turin, April 6–8, 1982 Proceedings*, chapter Specification and verification of concurrent systems in CESAR, pages 337–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[80] Yun-Bum Kim Radu Nicolescu, Michael J. Dinneen. Structured Modelling with Hyperdag P Systems: Part A. *Membrane Computing, Seventh Brainstorming Week, BWMC 2009*, 2:85107, 2009.

[81] Yun-Bum Kim Radu Nicolescu, Michael J. Dinneen. Structured Modelling with Hyperdag P Systems: Part B. *Centre for Discrete Mathematics and Theoretical Computer Science*, 373, 2009.

[82] Yun-Bum Kim Radu Nicolescu, Michael J. Dinneen. Towards structured modelling with hyperdag P systems. *International Journal of Computers, Communications and Control*, 5(2):224–237, 2010.

[83] Dana Scott. Toward a mathematical semantics for computer languages. 1971.

[84] Petr Sosík and Rudolf Freund. *Membrane Computing: International Workshop, WMC-CdeA 2002 Curtea de Arges, Romania, August 19–23, 2002 Revised Papers*, chapter P Systems without Priorities Are Computationally Universal, pages 400–409. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[85] John von Neumann. The General and Logical Theory of Automata. *Taub*, pages 288–328, 1961.