



Automatic Selection of Statistical Model Checkers for Analysis of Biological Models

by

Mehmet Emin BAKIR

Thesis submitted for the degree of
Doctor of Philosophy

The University of Sheffield
The Department of Computer Science

01.11.2017

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

"In the name of Allah, the Most Gracious, the Most Merciful."

الْحَمْدُ لِلَّهِ رَبِّ الْعَالَمِينَ

"All praises and thanks be to Allah, the Lord of **worlds**."

Dedicated to my family.

Declaration

This thesis is submitted in the form of the **Alternative Format Thesis** which allows composing thesis from academic publications, alongside with the traditional thesis sections.

I hereby certify that this thesis includes published papers of which I am a joint author. I have included a written statement from co-authors, and my supervisor Dr Mike Stannett has sighted email and other correspondence from all co-authors attesting to my contribution to the joint publications. The contribution statements are included in Appendix C.

Acknowledgments

First and foremost, all praises are to Allah, I am very grateful for blessing me with the opportunity and strength to undertake this research.

I would like to express my sincerest thanks to my supervisors, Dr Mike Stannett, Professor Marian Gheorghe, and Dr Savas Konur. I will appreciate forever for guiding me with your knowledge, arranging countless number of hours for meetings, and always supporting me throughout this study. I also want to thank my examiners, Dr Sara Kalvala and Professor Georg Struth for their suggestions, which helped me to improve my thesis.

I am truly grateful to all of my family members, especially my parents, for trusting and supporting me throughout my life. Last but definitely not least, I am feeling so lucky to have the infinite support of my wonderful wife, Ebru Bakir. Her encouragement, support, and wise advice illuminate my life.

This study would not have been possible without the PhD studentship provided by the Turkey Ministry of Education.

Abstract

Statistical Model Checking (SMC) blends the speed of simulation with the rigorous analytical capabilities of model checking, and its success has prompted researchers to implement a number of SMC tools whose availability provides flexibility and fine-tuned control over model analysis. However, each tool has its own practical limitations, and different tools have different requirements and performance characteristics. The performance of different tools may also depend on the specific features of the input model or the type of query to be verified. Consequently, choosing the most suitable tool for verifying any given model requires a significant degree of experience, and in most cases it is challenging to predict the right one.

The aim of our research has been to simplify the model checking process for researchers in biological systems modelling by simplifying and rationalising the model selection process. This has been achieved through delivery of the various key contributions listed below (p. vii; see also Sect. 1.3 for a more detailed discussion).

Contributions

- We have developed a software component for verification of kernel P (kP) system models, using the NuSMV model checker. We integrated it into a larger software platform (www.kpworkbench.org).
- We surveyed five popular SMC tools, comparing their modelling languages, external dependencies, expressibility of specification languages, and performance. To best of our knowledge, this is the first known attempt to categorise the performance of SMC tools based on the commonly used property specifications (property patterns) for model checking.
- We have proposed a set of model features which can be used for predicting the fastest SMC for biological model verification, and have shown, moreover, that the proposed features both reduce computation time and increase predictive power.
- We used machine learning algorithms for predicting the fastest SMC tool for verification of biological models, and have shown that this approach can successfully predict the fastest SMC tool with over 90% accuracy.
- We have developed a software tool, SMC Predictor, that predicts the fastest SMC tool for a given model and property query, and have made this freely available to the wider research community (www.smcpredictor.com). Our results show that using our methodology can generate significant savings in the amount of time and resources required for model verification.

- We have disseminated our findings to the wider research community in a co-authored collection of peer-reviewed conference and journal publications.

Contents

Title Page

Declaration i

Acknowledgments iii

Abstract v

Contributions vii

I Background to the Research 1

1 Introduction 2

1.1 General overview 2

1.2 Motivation and aims 4

1.3 Contributions 7

1.3.1 NuSMV translator 7

1.3.2 Comparative analysis of SMC tools 10

1.3.3 A set of model features for SMC performance prediction 11

1.3.4 SMC performance prediction 12

1.3.5 SMC Predictor tool 13

1.4	Structure of thesis	13
1.5	Publications used in this thesis	15
1.6	Publications arising from this research	18
2	Modelling	20
2.1	P systems	21
2.2	Kernel P (kP) systems	23
2.2.1	Preliminaries	24
2.2.2	kP systems definitions	24
2.2.3	kP systems rules	25
2.2.4	kP systems execution strategies	27
2.3	kP-Lingua	29
2.4	Stochastic P systems	31
2.5	Case Study - Gene expression	33
2.5.1	kP Systems (kP-Lingua)	34
2.5.2	Stochastic P Systems	34
2.6	Other formalisms	35
2.6.1	Petri nets	36
2.6.2	π -calculus	38
2.6.3	Mathematical models	39
2.6.4	Specification languages	41
3	Analysis	43
3.1	Model checking	43
3.2	Temporal logics	46
3.2.1	Linear-Time Temporal Logic (LTL)	47
3.2.2	Computational Tree Logic (CTL)	49
3.2.3	Probabilistic Computation Tree Logic (PCTL)	52
3.2.4	Continuous Stochastic Logic (CSL)	54

3.3	Statistical model checking	57
3.4	Model checking tools	60
3.5	Other analysis methods	62
3.5.1	Simulation	62
3.5.2	Testing	63
4	Automating The Model Checking Process	64
4.1	Integrated software suites	64
4.1.1	kPWorkbench	65
4.1.2	Infobiotics Workbench	74
4.2	Machine learning	75
II	The Papers	82
5	Modelling and Analysis	83
6	Comparative Analysis of Statistical Model Checking Tools	107
7	Automating Statistical Model Checking Tool Selection	127
III	Concluding Remarks	161
8	Achievements, Limitations and Future Research	162
8.1	Achievements	162
8.1.1	Objective 1: Summary of Progress	163
8.1.2	Objective 2: Summary of Progress	164
8.2	Limitations	166
8.2.1	NuSMV translator	166
8.2.2	SMC prediction	167
8.3	Future research directions	168

Appendices	171
A A Model Translated using the NuSMV Translator	171
B Instantiation of the property patterns	178
C Contribution Statements	180
References	185

List of Figures

1.1	SMC predictor architecture and work-flow (taken from [12]).	14
2.1	Virtuous circle between modelling and wet-lab experiments (adapted from [4]).	21
2.2	Stochastic Petri net model of gene expression (adapted from [19])	38
2.3	π -calculus model of gene expression (taken from [19]).	40
4.1	kPWorkbench Architecture (taken from [53]).	67
4.2	A sample of data for machine learning is an ordered pair of a row vector of features (input) and a target variable (output).	76
4.3	An overview of supervised machine learning algorithm.	77
4.4	k-fold cross-validation.	79

List of Tables

2.1	Mapping between biological systems and P systems	23
2.2	kP-Lingua model of the gene expression.	35
2.3	Stochastic P model of the gene expression, taken from [19]	36
2.4	Mapping between biological systems and Petri nets [19].	37
2.5	Mapping between biological systems and π -calculus [19]	39
4.1	The LTL and CTL property constructs currently supported by the kP- Queries file (taken from [53]).	66

Part I

Background to the Research

Chapter 1

Introduction

1.1 General overview

The structures and functionalities of biological systems, ranging from DNA to entire organisms, are extremely complex and dynamic. Their behaviours are hard to predict, because multiple components simultaneously perform an enormous number of reactions, and interact with other spatially distributed components under a wide variety of often unpredictable environmental conditions [11, 46]. Having a better understanding of the intra- and inter-component relations, and the working mechanisms of biological systems, would help us better understand the causes of diseases and develop effective treatments. Gaining insights into the complex behaviour of biological systems can also inspire researchers to infer new computational techniques and apply these methods widely across different scientific disciplines. Since, however, the capacity of the human brain alone is not enough to comprehensively understand the functionality and complexity of biological systems, researchers have increasingly turned to machine-executable models.

These models provide an abstract representation of the essential parts of real systems, such as living cells. They usually capture elements of interest at different levels of abstraction (rather than the system as a whole), and neglect less important details whenever possible [28, 60, 81]. Computational models can be used, in particular, to provide insights

into the fine-grained spatial and temporal dynamics of biological systems. The machine-readable and executable attributes of computational models also empower researchers to carry out *in silico* experiments, including hypothesis testing, in a faster, cheaper, and more repeatable way than the corresponding *in vivo* and *in vitro* experiments [11, 12].

It is important, of course, that models are representative, i.e. they should accurately represent the aspects of the modelled system that are of interest to researchers. In the context of the bio-computational models discussed in this thesis, two powerful and widely used approaches to validation and verification are *simulation* and *model checking*. Other approaches are also possible, for example, testing (see Section 3.5.2).

- **Simulation** works by generating and analysing a representative set of execution paths. It can generate paths relatively quickly, but does not guarantee the exhaustive generation of all possible paths, especially in large non-deterministic models [11].
- **Model checking** demonstrates formally that a model exhibits specified properties by exhaustively investigating its entire state space [7, 35, 64]. As this method guarantees the correctness of the specified property by this exhaustive exploration, it is called *exact model checking*. However, the exhaustive analysis suffers from the well-known state-space explosion problem, i.e. the state space to be analysed grows exponentially as the size of the model increases, which means that model checking techniques can only be used for the verification of relatively small models. Model checking has been used extensively throughout this study; we provide a formal outline in Section 3.1, and discuss practical aspects in the published papers in Part II.

To mitigate the state explosion problem associated with the exact model checking and so enhance scalability, a less resource-intensive *statistical model checking* (SMC) approach may be employed, which attempts to combine the speed of simulation with the intensive analytical capabilities of model checking. Instead of examining the entire state space, SMC investigates only a subset of the possible execution paths, and performs model checking through approximation (this approach has many parallels with software testing, and similar concerns as to coverage and integration apply (see Section 3.5.2)). The considerably

decreased number of execution paths allows the verification of much larger models with less computational resources, but does so by introducing a small amount of uncertainty [12].

These powerful features of model checking techniques have led researchers to implement new model checking tools for use across a variety of fields [48, 70, 71, 103]. However, this piecemeal approach means that the requirements for one tool can be different to those for another [11]. For example, different tools may employ different modelling and specification languages. Thus, to use a combination of different model checkers, users must first be familiar with a range of different model checker languages and know how to use the associated checkers—this can be a cumbersome task, especially for those biologists who are not already experts in the methods involved.

To streamline the model checking process, various integrated software suites have been developed. Of particular relevance for our purpose is the *kPWorkbench* [41, 72], but some other frameworks also exist, for example, *Infobiotics Workbench* [24, 25] (see Section 4). These tools enable the integration of different simulation and model checking tools on a single platform, and simplify model checking by providing high-level modelling and property specification languages—these are internally translated into the formats required by the user’s selected model checker, and the corresponding verification process can be automatically initiated [12]. Nonetheless, these integrated suites still rely on the user’s expertise in selecting the most appropriate model checker. Users need to know which model checker is best for their particular model and the properties to be checked, and this in itself requires significant verification experience [12].

1.2 Motivation and aims

The existence of multiple variants of statistical model checking tools gives flexibility and allows users to perform fine-tuned analysis of models. However, it comes at the cost of lack of clarity as to which tool is best for verifying any particular model, because each tool has its own practical limitations, and different tools have different requirements and performance characteristics. Moreover, the best tool for analysing any given model may also depend

on other factors, such as the queried property type or the population of molecules. Thus, users must consider a wide range of trade-offs and drawbacks when selecting the most appropriate SMC tools for the particular scenarios in which they are interested. This is not an easy task and requires a significant degree of verification experience, because [12]:

- Each tool typically employs different modelling and property specification languages, and tools differ as to which properties can be expressed. That is, a property which is easy to express and verify using one tool may not be expressible at all using another tool. In order to verify multiple properties, users may therefore need to learn a range of model checking technologies, as well as multiple modelling and property specification languages.
- Some tools are self-contained, while others rely on external third-party tools to carry out pre-processing of the models to be investigated, which means that users need to become acquainted with the external tools as well.
- The performance of different tools may vary significantly depending on the characteristics of the input model and the property type. While one tool may verify certain models and properties efficiently, another may require more computational power and time for the same task, or even fail to complete the verification altogether. In one particular scenario, an SMC tool may run for hours or days to complete its task while simultaneously taking control of almost all of the available resources, whereas another SMC tool may be able to accomplish the same task both faster and using fewer resources; but conversely, in another scenario, the first tool may accomplish its task faster than the second one.

Therefore, identifying the most suitable SMC tool is an error-prone, cumbersome, and time-consuming process. Given the difficulties associated with using multiple model checkers, it is currently common practice to verify models using whichever SMC the user is most familiar with, whereas another tool might actually be more effective, for example, the time differences between the default and the actual best one can be orders of magnitude. These

complications can dissuade researchers from analysing complex models or verifying fine-grained properties. We argue, therefore, that it is desirable to minimise human intervention by developing techniques for automatically identifying the most suitable SMC tool for a given biological model and property specification prior to the verification [11]. Automating tool selection in this way should, we believe, significantly decrease the consumed cost, time, and effort for model verification. As a result, it should enable easier verification of complex models, and thereby lead to a better and deeper understanding of biological systems [12].

As an alternative approach, we could run all SMC tools in parallel, and once the fastest one completes its execution, it could be made to raise a flag so that we can halt the remaining running tools. However, as SMC tools are resource intensive, if we run several of them in parallel each of them will demand a significant amount of resources, and taken together we risk blocking all available resources throughout the execution. Blocking so many computational resources will potentially increase the individual verification time of each tool, which means that even the fastest tool would run slower than if it had run alone. The use of so many unnecessary resources also increases environmental impact in terms of energy consumption. In contrast, by using our approach of automatic fastest tool selection we can rapidly decide the best solution and use just a single SMC tool, thereby saving significant time and resources which can be used for other purposes.

Aim and objectives. Our fundamental aim is to simplify the model checking process for researchers in biological systems modelling.

Our *initial objective* was to investigate the use of exact model checking for verifying biological models; a key part of this approach was the development of a software component that can map a high-level modelling language to an exact model checker language (see Section 4.1.1.1). It became apparent, however, that the exact model checking approach is unable to cope with the demands of larger models.

We accordingly shifted focus to look at statistical model checkers, since these are well known to be more capable of dealing with large models. However, even here there were shortcomings since different tools had different performance characteristics and it was not

possible to know with any certainty which SMC would be best for each combination of model and query. This informed our *second objective*: to develop a system which can predict, for the verification of any given model and property combination, which SMC is the fastest choice.

1.3 Contributions

1.3.1 NuSMV translator

kPWorkbench (see Section 4.1.1) is an integrated software framework which aims to provide a set of tools to help researchers in their modelling and analysis of kP systems (these are a special type of P systems, a general computational model inspired by the structure and functionality of living cells [41, 72]). For further details, including the formal definition of kP systems, see Section 2.2.

Initially, kPWorkbench had support only for the Spin model checker [113], in that it included a component (the *Spin translator*) for translating kP system models expressed using its domain specific language, kP-Lingua (see Section 2.3), into Spin model checker specifications expressed in Promela (Process Meta Language) [113]. However, the way in which Spin builds entire state spaces and performs searching limits it to verifying only very small kP system models. Inevitably, therefore, the Spin translator component adopted a strategy to restrict the generation of too large a state space, and did this by adding an upper bound variable to the Spin specification, which restricts the number of computation steps permitted. By design, however, this means that only part of a state space is generated and investigated. This is a fundamental issue, because these restrictions mean that we no longer have full confidence in the results of the ensuing model checking. Another restriction stemming from the use of Spin is that it only allows the verification of model properties that can be expressed using Linear-Time Temporal Logic (LTL) (a formalism for specifying model features; see Section 3.2).

To avoid these issues relating to the Spin translator, I decided instead to employ the

NUSMV model checker [79]. NUSMV is a symbolic model checker, which means it does not construct individual states explicitly; rather, it symbolically represents sets of states by using ordered binary decision diagrams, which allows the representation of larger state spaces in a compressed form [64]. In addition to LTL, NUSMV supports Computational Tree Logic (CTL), which is another formalism for querying model properties (again for more information on CTL, see Section 3.2). To support my work, I have accordingly developed a software component – now integrated into kPWorkbench – which can translate models specified in kP-Lingua into NUSMV specification language. Performing this translation is not a simple and straightforward process; indeed, we had to address a number of interesting challenges:

- One challenge is inherent in the model checking strategy itself. Because the size of the state space is one of the primary problems of the model checking method, the translation process should carefully construct conditional rules that determine the possible values of constructed variables. For example, kP system objects do not have implicit bounds, but we nonetheless need to identify upper and lower bounds for each of the corresponding NUSMV variables, and also include options to explicitly change those bounds. The state space of the translated model eventually determines the feasibility of the verification, the translation procedure is explained in Section 4.1.1.1.
- Next, there were challenges associated with NUSMV model checker specifications: NUSMV allows defining arrays, but (unlike Promela) it does not allow using a variable as an index for accessing or assigning arrays values. Additionally, the array size has to be a constant and cannot be changed after declaration. This is unfortunate, as the structuring of a kP system’s compartments resembles a graph which can be stored using two dimensional arrays, provided these can be modified – so the lack of array support makes it hard to achieve a complete translation from kP systems to NUSMV specifications (see Section 4.1.1.1 for more information concerning these translation limitations). Another challenge that stemmed from the nature of NUSMV

specification is the lack of non-determinism, because most kP system rules are executed non-deterministically, whereas NUSMV always deterministically selects the first applicable rule. In some model checkers (for example, Spin), non-deterministic selection is internally enabled, but for NUSMV I had to find a workaround to achieve this.¹

- Finally, there was the challenge posed by the complexity of the structure and the behaviour of kP systems. The kP systems formalism not only allows the formation of complex structures, e.g. the network structure linking compartments, but also allows these structures to be modified dynamically. Moreover, some kP systems rules are expected to run in parallel, whereas NUSMV runs rules sequentially. To overcome these issues, I had to design the translated model so that NUSMV runs sequentially but presents apparently parallel behaviour.

These constraints made it very difficult to achieve a complete translation from kP systems to NUSMV specifications. It became possible only after introducing and orchestrating numerous new rules and variables—for example, the kP system model of Example 2 on page 31 has fewer than 30 lines of kP-Lingua code, but its translated NUSMV model requires more than 400 lines.² Currently, the NUSMV translator is the largest component of the kP-Workbench in terms of lines of code, and it has successfully been used to verify a number of models from different fields, including biology and engineering; see, e.g., [9, 51, 53, 54, 74]. We always update the website that we have built for kPWorkbench.³ It internally accommodates the latest version of the NUSMV translator. The website also includes several case studies which are modelled in kP-Lingua and their translated NUSMV models.

¹Roughly speaking, whenever a non-deterministic behaviour is required, we have to combine the set of applicable rules and exclude the default true rule (a default rule is required by the NUSMV specification [34]). This raises another issue, in that NUSMV supports the set union operation for rules, but it does not have a set difference operation to exclude the default true rule. I used NUSMV invariants (which are propositional formulas that hold invariantly [79]) to assert that if any rule is applicable then the default rule is not applicable.

²These models are available online at: <http://www.github.com/meminbakir/kernelP-store>.

³<http://www.kpworkbench.org>

1.3.2 Comparative analysis of SMC tools

Selecting the most suitable SMC tool for a given task is not easy, especially for non-expert users (including typical biologists). Our work ([11]) is meant to help guide researchers in choosing proper SMC tools for biological model verification, without the need for any third-party application. To this end I surveyed five popular SMC tools by comparing their modelling languages, external dependencies, expressibility of specification languages, and performances on verification of 475 biological models and 5 property patterns. To the best of our knowledge, this is the first attempt to categorise the performance of SMC tools based on the commonly used property specifications (commonly called **property patterns**, or just **patterns** [42, 58]).

This study resulted in two main contributions of knowledge to the field. First, the study identified the boundaries where the fastest model checker can be determined by examining just the model size and the property pattern. For example, when the model size is very small or large, users can quickly decide which model checker will be the fastest tool for their experiments, based simply on model size and the property pattern. We showed, however, that there are cases where the fastest tool cannot be determined by examining these two parameters alone, and that a more extensive investigation is required. Second, the study made clear for the first time the relationship between property patterns and the performance of SMC tools, by showing that the type of queried properties remarkably affects the total verification time. These findings convinced us that we should group the performance of the tools on a per-pattern basis, and take the property pattern type into account when predicting the fastest tool. Later, I extended this study substantially by considering a much broader set of models (675 models in total) and property patterns (11 patterns); this more comprehensive experiment and its prediction results are reported in Section 7.

The benchmarking of overall performance was very time consuming. We ran each tool three times for the verification of each model and the property pattern. Although we limited each run to a maximum verification time of one hour, this still gave a worst-case

scenario of requiring 111,375 hours' total verification time (i.e. more than 12.7 years).⁴ Fortunately, however, smaller models were verified faster, and using my default computer settings (see Section 7) I managed to complete all executions within 5-6 months—while considerably shorter than the worst-case scenario, this remains a significant amount of time. These performance results are publicly available online at <http://www.smcpredictor.com/data.html>.

1.3.3 A set of model features for SMC performance prediction

The features used for prediction are typically crucial for determining the success of projects that rely on machine learning algorithms [39]. In the context of predicting the fastest SMC, we need to determine which model features most influence the performance of the various tools; this is vital for correctly predicting the fastest SMC. Identifying these features is relatively easy when only a small number of parameters are relevant for prediction and the problem domain is well known. In such cases, we can simply program a typical algorithm for the task and machine learning is unnecessary. For more complex cases, when many factors determine the outcome that we want to predict, it may no longer be intuitively obvious how to identify the right number of features, and machine learning can be useful.

As I mentioned earlier, the type of property being queried is one factor that affects the performance of SMC tools [11]. In related work [12] I have shown that the characteristics of the input models, such as the number of species and reactions, also affect the performance of the tools. I therefore investigated 12 new model features for predicting the fastest SMC tool for verification of biological models, and showed that using this new feature set increases predictive accuracy (the ratio of correct predictions (true positives) to total sample size) and yields algorithms that are fast to compute; see Chapter 7, Figure 2. Additionally, I conducted comparative experiments that demonstrated that the proposed features can also improve on previously reported accuracy of predicting the fastest Stochastic Simulation Algorithms (SSAs) used for simulating biological models.

⁴The time required for the worst-case scenario is given by: 5 SMCs \times 675 models \times 11 property patterns \times 3 executions per combination \times 1 hour per execution = 111,375 hours.

1.3.4 SMC performance prediction

In summary, predicting the fastest SMC tool requires us to consider more features than just the model size and property patterns involved. However, when the number of the model features increases, it becomes harder to formulate the relationship between those features and system performance. Machine learning algorithms are suitable candidates (sometimes the only viable choice) for such problems (see Section 4.2)—for example, supervised machine learning algorithms can gradually learn from data how to map inputs to desired outputs. So-called *regression* algorithms are used if the desired output belongs to a continuous distribution, while for discrete distributions (or categorical variables) *classification* algorithms are more appropriate. In our case, for predicting the fastest SMC, the desired output is a categorical variable (its value can be one of five candidate model checkers), so we have focused our attention only on classification algorithms.⁵

After benchmarking the performance of SMC tools against different property patterns and identifying the model features which are important for performance prediction, I used the property patterns and the model features as input for five machine learning algorithms that are known to be appropriate for classification problems, and used them to predict the fastest SMC tool. I compared the accuracy of each algorithm for different property patterns. The experiment details and the results are reported in Section 7. The results showed that Extremely Randomized Trees (ERTs) [50] were the best classifier for six property patterns, while Support Vector Machines (SVMs) [33] were best for the other five property patterns. The best classifier for each property pattern type could predict the fastest SMC tool with over 90% accuracy. I also demonstrated that using the best classifiers can save users a significant amount of time—up to 208 hours!

⁵Note, our goal was not the construction of new classification algorithms, but the use of existing classification techniques in constructing a prediction algorithm. The classification algorithms we used are all well-known in the community.

1.3.5 SMC Predictor tool

SMC Predictor is a standalone application designed to automate the prediction of the fastest SMC tools for a given model and property query. Figure 1.1 shows the tool architecture and demonstrates its work-flow. The tool accepts a biological model in the Systems Biology Markup Language (SBML) format, an XML based format for describing and exchanging biological processes [63], and a property pattern file specified in the Pattern Query Language (PQL) ⁶, a high-level domain specific language for formulating model properties with natural-like keywords [82]. SMC Predictor modifies the SBML model, such as by removing multiple compartments and fixing the population of species, then delivers the modified model to Model Topological Analysis component. The Model Topological Analysis component translates the model to species and reaction graphs and extracts graph-related features (e.g. number of vertices and edges, graph density), and non-graph features (e.g. number of non-constant species, number of species multiplied by number of reaction) of the model (see Chapter 7). The Predictor component accommodates the pre-trained machine learning algorithms, it takes the topological features of the model and property pattern and delivers them to the machine learning algorithm. The machine learning algorithm evaluates the model features and predicts the best SMC tool accordingly. More detail on the prediction and the SMC Predictor tool are provided in Chapter 7, and its dedicated web site www.smcpredictor.com. Its code is also publicly available and free to use, on www.github.com/meminbakir/smcp.

1.4 Structure of thesis

This thesis structure follows the Alternative Format Thesis scheme. The alternative format thesis scheme allows composing chapters from academic publications, alongside with the traditional thesis sections. Part II composed of some academic papers.

Part I outlines the background for modelling, analysis and automating the model

⁶PQL grammar can be accessed from www.smcpredictor.com/pqGrammar.html

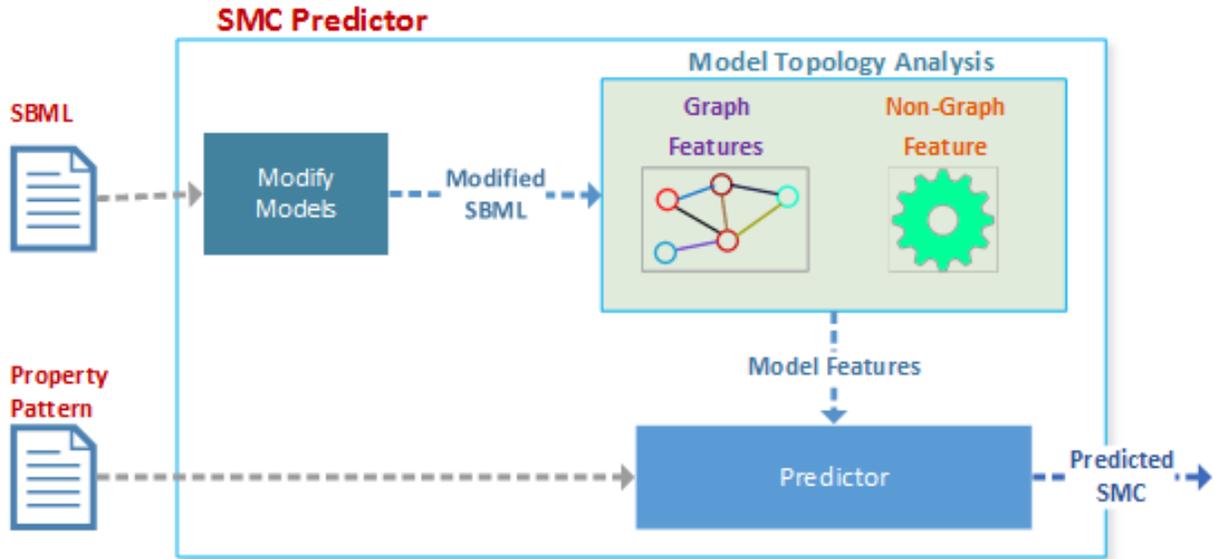


Figure 1.1: SMC predictor architecture and work-flow (taken from [12]).

checking, which covers the overview of: *(i)* modelling concept, different modelling formalisms (particularly kernel P systems) which are used for describing biological systems; *(ii)* a biomolecular system example for concretizing some of the computational modelling formalisms; *(iii)* the analysis methods used for investigating model properties, especially the property specification languages, some of the model checking tools, and simulation and testing as alternative analysis methods; *(iv)* and the integrated software suites which integrate modelling, simulation and model checking of biological systems on a single platform, it especially focuses on the NUSMV translator. *(v)* machine learning, and how it is used for predicting the fastest SMC tool.

Part II binds together our various publications used to support this thesis, and provides a narrative explaining how the work in each paper leads on to the next. The summaries of the papers are in the next section.

Part III comprises a single chapter (Chapter 8), in which we summarise our research, present our conclusions, and suggest possible avenues for further research.

1.5 Publications used in this thesis

Chapter 5, 6, and 7 (in Part II) are in academic publication format. Although each paper in these chapters represents novel results, the papers investigate similar fields. Therefore, some degree of duplication exists, particularly in the background sections.

Chapter 5 presents two conference papers which focus on computational models, and their analysis with model checking and simulation. The papers show how the simulation and exact model checking approaches complement each other, thereby providing a better understanding of the functionality of systems.

First paper: Mehmet E. Bakir, Florentin Ipate, Savas Konur, Laurentiu Mierla, and Ionut Niculescu. Extended simulation and verification platform for kernel P systems. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *Membrane Computing: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers*, pages 158–178. Springer International Publishing, Cham, 2014.

This paper presents two computational modelling formalisms and their analysis using model checking and simulation methods. It begins by describing and formally defining two types of P system used for modelling, namely kernel P (kP) systems and stochastic P systems, as well as two finite state machine extensions, namely the stream X-machine (SXM) and communicating stream X-machine (CSXM), which constitute the formal basis of the FLAME (Flexible Large-scale Agent Modelling Environment) simulator. This leads into our presentation of two novel extensions to the kPWorkbench software framework: a formal verification tool based on the NUSMV model checker, called NUSMV translator (see Section 4.1.1.1), and a large-scale simulation environment based on FLAME.

In this paper we model an example biological system as both a kP system and as a stochastic P system, and then show how to analyse it using model checking and simulation components of kPWorkbench. Analysis with model checking was not immediately possible for the original model, with its many rules and compartments, because of the state explosion problem. The paper therefore introduces a compact representation (with

fewer compartments and rules) of the model, which is more suitable for verification, and it also demonstrates the use of simulation as a complementary analysis technique. We show that the compact model reproduces the same behaviour as the original one. Therefore, although model checking is applicable only for relatively small models, our compactification technique can help to verify larger systems.

Second paper: M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High performance simulations of kernel P systems. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICSS)*, pages 409–412, 2014.

The first study showed that when the model size is large, we cannot use model checking without compactification, so that simulation is more appropriate in this situation—we accordingly investigated how to use two simulation tools, kPWorkbench Simulator [8] and FLAME [37] to analyse a sample biological system. In this second study, we compared the performance of these two simulators (with default settings) by executing them on a large synthetic biology model coded as a kP system. The results highlight the performance difference between using a general-purpose simulator (FLAME) as opposed to a custom kP systems simulator (kPWorkbench Simulator).

Chapter 6 (Third Paper): Mehmet Emin Bakir, Marian Gheorghe, Savas Konur, and Mike Stannett. Comparative analysis of statistical model checking tools. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing: 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers*, pages 119–135. Springer International Publishing, Cham, 2017.

In the previous studies, we focused on expressing biological systems as computational models and analysing them using both simulation and model checking techniques. However, model checking can be applied either exhaustively or approximately. Since exact model checking suffers from the state explosion problem, it can be used in practice only on small

models. On the other hand, statistical model checking blends the simulation and model checking approaches by considering a fraction of simulation traces, rather than all traces, and provides approximate correctness of a queried property. Typically, statistical model checkers can verify larger models, and they are faster than exact model checkers. These advantages of statistical model checking have persuaded scientists to develop a number of tools to implement this formalism. Although the diversity of the tools gives flexibility to users, it is not clear which statistical model checking tool is going to be fastest for a given model and property pattern.

In this third study we reviewed five statistical model checking tools by comparing their modelling and specification languages, as well as the property patterns that they support. We also examined their usability regarding expressibility of property specification, and benchmarked the performance of the tools on verification of 465 biological models against five property patterns. The experiments showed that the performance of different tools significantly changes for different property pattern types, and also as model size changes. We found that in some cases the best model checker can be identified by considering only the model size and the property pattern, and for those cases we provide guidance for those scenarios where a user can spot the fastest SMC tool without using any third-party tool. The study also identified boundaries where the best choice is less clear-cut. For such cases, examining the model size and property pattern parameters alone is not enough, and the fastest SMC tool cannot be identified intuitively. The findings point to a clear need for automating the SMC tool selection process, as well as the need to explore additional model features that might be relevant to simplifying this task.

Chapter 7 (Fourth Paper): Mehmet Emin Bakir, Savas Konur, Marian Gheorghe, Natalio Krasnogor, and Mike Stannett. Performance benchmarking and automatic selection of verification tools for efficient analysis of biological models. Unpublished, 2017.

This unpublished paper presents an initial, and in some respects more comprehensive, account of work subsequently accepted (after formal submission of this thesis) for publication in revised form as [13].

In this study, we proposed and implemented a methodology for automatically predicting the most efficient SMC tool for any given model and property pattern. Machine learning algorithms usually generate better results if they have more data, so as part of this study we significantly extended the data set used in our previous paper [11] by performance benchmarking a much larger set of biological models and verifying them against more property patterns. In all, we recorded the performance of 5 SMCs for 675 models verified against 11 property patterns. While our earlier work aimed to help users manually identify the most appropriate SMC tool whenever possible, this study focuses on how to **automate** the model checker selection process, and the efficiency of our approach.

To match the best SMC tool to a given model we proposed a novel extended set of model features, and showed that the proposed features are computationally cheap and important for prediction of the fastest SMC tools and SSAs. Using several machine learning algorithms, we were able to successfully predict the fastest SMC tool with over 90% accuracy for all pattern types. We developed a software utility tool, **SMC Predictor**, which predicts the fastest SMC tool for verification of biological models. Finally, we demonstrated that using our SMC predictor tool provides real benefits to users in terms of time savings.

1.6 Publications arising from this research

1. Mehmet Emin Bakir, Savas Konur, Marian Gheorghe, Natalio Krasnogor, and Mike Stannett. Automatic selection of verification tools for efficient analysis of biochemical models. *Bioinformatics*, 2018. Available online: <https://academic.oup.com/bioinformatics/advance-article/doi/10.1093/bioinformatics/bty282/4983061>.
2. Raluca Lefticaru, Mehmet Emin Bakir, Savas Konur, Mike Stannett, and Florentin Ipatu. Modelling and validating an engineering application in kernel P systems. In Marian Gheorghe, Savas Konur, and Raluca Lefticaru, editors, *Proceedings of the 18th International Conference on Membrane Computing (CMC18)*, pages 205–217. University of Bradford, July 2017

3. Mehmet Emin Bakir, Marian Gheorghe, Savas Konur, and Mike Stannett. Comparative analysis of statistical model checking tools. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing: 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers*, pages 119–135. Springer International Publishing, Cham, 2017
4. Mehmet E. Bakir and Mike Stannett. Selection criteria for statistical model checking. In M Gheorghe and S Konur, editors, *Proceedings of the Workshop on Membrane Computing WMC 2016, Manchester (UK), 11-15 July 2016*, pages 55–57, 2016. Available as: Technical Report UB-20160819-1, University of Bradford
5. Marian Gheorghe, Savas Konur, Florentin Ipate, Laurentiu Mierla, Mehmet E. Bakir, and Mike Stannett. An integrated model checking toolset for kernel P systems. In *Membrane Computing: 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers*, pages 153–170. Springer International Publishing, Cham, 2015
6. M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High performance simulations of kernel P systems. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on CyberSpace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICSS)*, pages 409–412, 2014
7. Mehmet E. Bakir, Florentin Ipate, Savas Konur, Laurentiu Mierla, and Ionut Niculescu. Extended simulation and verification platform for kernel P systems. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *Membrane Computing: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers*, pages 158–178. Springer International Publishing, Cham, 2014

Chapter 2

Modelling

In order to understand the structure and working mechanism of biological systems, for years, biologists have utilised diagrammatic models, which could only provide very limited information [46, 94]. However, to gain comprehensive insights into such systems, more advanced techniques, which can highlight the spatial and temporal evolution of the systems, are required. Scientists start to harness computers to get a better and deeper understanding of the spatial and time-dependent behaviour of biological systems [46]. Therefore, machine-executable mathematical and computational models and specification languages for biological systems are developed.

Models are an abstract representation of the real-systems, such as living cells. They usually capture the essence of the interest in the different levels of abstraction, but not the whole system, and neglect less critical details whenever possible [28, 60, 81]. The machine-readable and executable attributes of models empower researchers to carry out *in silico* experiments in a faster, cheaper, and more repeatable way than the similar wet-lab experiments [11]. Models are useful for hypothesis testing, cause-and-effect chain tracing, and for gaining new insights [29]. The verified models help to redesign the wet-lab experiments and inspire further tests. The virtuous circle between modelling and wet-lab settings enables incrementally getting a better understanding of biological phenomena, see

Figure 2.1. Furthermore, models permit investigating systems that are difficult to explore in real-world settings, due to practical constraints, such as cost, size, and ethics [4].

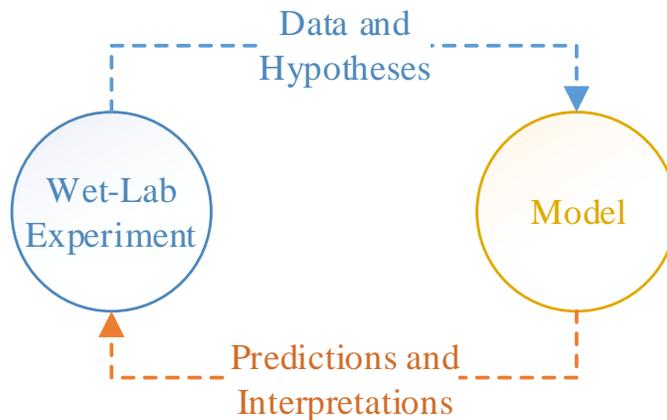


Figure 2.1: Virtuous circle between modelling and wet-lab experiments (adapted from [4]).

Key. Wet-lab experiment proposes a new hypothesis which can be specified and verified with the models. The model will either prove or refute the hypothesis, and it can help to generate further predictions, which can be used as input for the next iteration of wet-lab experiments.

Models can be described in equation-based (e.g., ordinary differential equations), graphical (e.g., Petri nets, statecharts) or textual (e.g., rule-based system) representation [15]. They can represent either continuous, discrete, or hybrid systems, and can be executed deterministically (fixed execution path) or stochastically (random execution path) [4]. In the following sections, we summarise some of these methods and we present some examples to explain them further.

2.1 P systems

Membrane Computing [65, 84] is a branch of Natural Computing inspired by structure and functioning of living cells. It focuses in particular to the features that originate from the presence and functionality of membranes. The computational models used for describing

membrane computing are called **P systems**, which are, in general, distributed and parallel. P systems can be categorized as **(i)** cell-like P systems which imitate the hierarchical and selectively permeable membranes (eukaryotic cells) and form tree-like structures, **(ii)** tissue-like P systems have a colony of one-membrane cells living in the same environment, all cells can communicate via environment, only specific cells can communicate each other, and they form a graph-like structure, and **(iii)** neural-like P systems, which are inspired by the neurons interacting in neural nets, they are similar to tissue-like P systems, in addition, they have a state which controls the system evolution [65, 84, 85]. For a formal account of general P system models (including standard mathematical concepts like strings, multisets, and the encoding of transition rules as rewrite systems) we refer the reader to [83]. Our focus in these chapters is the particular variant known as **kernel P systems**, which is introduced in detail in the following section.

P systems consist informally of membrane structures (each membrane can contain zero or more further membranes), multisets of objects (multiple instances of the same object), which are placed in the compartments delimited by the membranes, and a set of rules for processing objects and membranes as time passes [65, 70, 71, 83, 85]. Various execution strategies can be used to define the behaviour of compartment types, that is, how the rules will be applied. The application of rules in a P system is traditionally done in a maximally parallel way, which means that at each time step a (typically randomly selected) maximal collection of applicable rules will be applied simultaneously, and then the system moves to the next time step.

For computational purposes, the initial distribution of objects represents the system input, and the final distribution (more specifically the number of objects in some pre-selected compartment) represents the corresponding output. From the computational point of view, most variants of the basic P system model can simulate arbitrary Turing machines, that is, they are computationally complete (Turing complete) [85], and much work has been carried out to find the number of membranes in a P system that are sufficient to characterise the power of Turing machines [65, 84, 85]. P systems are also computationally efficient because they are inherently parallel computing devices, that is, all membranes

Biological Systems	P Systems
Compartment	Membranes
Molecule	Objects
Molecular population	Multisets of objects
Biochemical transformation	Rules
Compartment translocation	Rules

Table 2.1: Mapping between biological systems and P systems

and their objects evolve simultaneously. This feature is mainly achieved by membrane division and membrane creation operations. However, there remains a trade-off between time and space, with space requirements for some models often increasing exponentially in linear time; this makes it possible to get polynomial-time (even linear-time) solutions to NP-complete problems [41, 65, 84, 85].

Membrane computing has been applied to a wide range of fields from biology, linguistics, economics, to computer science (in devising sorting and ranking algorithms), and cryptography [41, 65, 68, 84, 85]. It has been developed at various levels, with the following dimensions: (a) the newly introduced concepts, or a new manner in this area, (b) the mathematical formalism of membrane computing, and (c) the graphical language, the way to represent membrane structures, together with the contents of the compartments and the associated rules. [65, 85].

2.2 Kernel P (kP) systems

In this thesis, our focus is on a relatively recently introduced variant of the P system model, namely **kernel P (kP) systems** [41, 52]. kP systems encapsulate features of many other P system variants, but whereas standard P systems generally adopt maximal parallelism, kP systems allow a choice of different execution strategies for defining how the rules will be applied. We now provide a formal account of kP systems; in general our definitions and terminology follow those of [51, 52].

2.2.1 Preliminaries

A string over a finite alphabet A , where $A = \{a_1, \dots, a_p\}$, is a sequence of symbols from A . A^* denotes the set of all strings over A , λ denotes the empty string, and $A^+ = A \setminus \{\lambda\}$ denotes the set of non-empty strings. The length of a string u , where $u \in A^*$, is denoted by $|u|$, and $|u|_a$ denotes the number of $a \in A$ in u . For a subset $S \subseteq A$, $|u|_S$ denotes the number of occurrences of the symbols from S in u . The length of a string u is given by $\sum_{a_i \in A} |u|_{a_i}$. The length of the empty string is 0, i.e. $|\lambda| = 0$. A multiset over A is a mapping $f : A \rightarrow \mathbb{N}$. For an element $a \in A$, $f(a)$ is the multiplicity of a in f . The *support* of f is defined as $supp(f) = \{a \in A | f(a) > 0\}$. For $supp(f)$, the multiset is represented as a string $a^{f(a_{i_1})} \dots a^{f(a_{i_p})}$, where the order is not important.

Molecules in cellular systems can be represented by the multiset of objects in the kP systems. Objects in the multisets symbolise different molecules, and the multiplicity of objects represents the population of the corresponding molecule types.

2.2.2 kP systems definitions

Let's begin with compartment types which will be used for defining the compartments of the kP systems.

Definition 1 T is a finite set of compartment types, $T = \{t_1, \dots, t_s\}$, where $t_i = (R_i, \sigma_i)$, $1 \leq i \leq s$, consists of a set of rules, R_i , and an execution strategy, σ_i , defined over $Lab(R_i)$, the labels of the rule R_i .

The compartments of the kP systems are constructed from the compartment types. Each compartment, C , is a tuple (t, w) , where the compartment type $t \in T$ and w is its initial multiset. The rules and the execution strategies are discussed later.

In biological cells, molecular interactions can be delimited by membranes. The functionality of different membranes can be different for different organisms, cells or cell regions. Different membranes are specified as compartment type (see Definition 1) in kP systems

which can have its own strings (molecules), rules and execution strategies (molecular interaction) [19].

Definition 2 *A kP ($k\Pi$) system of degree n is a tuple*

$$k\Pi = (A, \mu, C_1, \dots, C_n, i_0) \quad (2.1)$$

- *where A is a finite set of elements, called objects.*
- *μ defines the initial membrane structure of the kP system. μ is an undirected graph, (V, E) , where V are vertices that signify the compartments, and E are edges that indicate the links between the compartments.*
- *$C_i = (t_i, w_i)$, $1 \leq i \leq n$, is a compartment of the system consisting of a compartment type from T , $t_i \in T$ (see Definition 1), and an initial multiset, w_i over A .*
- *i_0 is the output compartment where the result is received.*

2.2.3 kP systems rules

The kP system rules may have a guard g , the standard form of a rule is $r \{g\}$, where r is the rule, and g is its guard. The guards are constructed using multisets over A , as operands, together with the relational or Boolean operators, as explained below.

For a multiset w over A and an element $a \in A$, we denote by $|w|_a$ the number of objects a occurring in w . Let us denote $Rel = \{<, \leq, =, \neq, \geq, >\}$, the set of relational operators, $\gamma \in Rel$, a relational operator, and a^n a multiset with multiplicity n and $r \{g\}$ a rule with guard g . Let's first introduce the abstract relational expression.

Definition 3 *If g is the abstract relational expression denoting γa^n and w is the current multiset, then the guard g applied to w denotes the relational expression $|w|_a \gamma n$.*

The abstract relational expression g is true for the multiset w , if $|w|_a \gamma n$ is true.

An abstract Boolean expression is defined by one of the following conditions:

- any abstract relational expression is an abstract Boolean expression.
- If g and h are abstract Boolean expressions then $\neg g$, $g \wedge h$ and $g \vee h$ are abstract Boolean expressions, where \neg is negation, \wedge is conjunction and \vee is disjunction Boolean operators, in decreasing precedence order.

Definition 4 *If g is an abstract Boolean expression containing g_i , $1 \leq i \leq q$ where $q \in \mathbb{N}$, abstract relational expressions and w is a multiset, then g applied to w means the Boolean expression obtained from g by applying g_i to w for any i .*

The guard g is true for the multiset w , if the abstract Boolean expression g applied to w is true.

Example 1 *For the abstract Boolean expression $> k \wedge \geq 4l \vee \neg > 2m$ defined the guard g and its multiset w , then g applied to w is true if the multiset contains more than 1 k 's and at most 4 l 's or no more than 2 m 's.*

Definition 5 *A rule of a compartment $C_{t_i} = (t_i, w_i)$ can either have the type of a rewriting rule, that of a communication rule, or that of structure changing rule:*

- **(a) rewriting and communication rules:** $x \rightarrow y \{g\}$, where $x \in A^+$ and y has the form $y = (a_1, t_1) \dots (a_h, t_h)$, $h \geq 0$, $a_j \in A$ and t_j , $1 \leq j \leq h$, is a compartment type from T (see Definition 1) of compartments linked to the current one; t_j can be the same type of the current compartment, C_{t_i} ; if a link does not exist (i.e. there is no edge between the two compartments in E) then the rule is not applied; if a target, t_j , refers to a compartment type that has more than one compartments linked to C_{t_i} , then one of them will non-deterministically be selected.
- **(b) structure changing rules:** the following types of rules are considered:
 - **(b1) membrane division rule:** $[x]_{t_i} \rightarrow [y_1]_{t_{i_1}} \dots [y_p]_{t_{i_p}} \{g\}$, where $x \in A^+$ and $y_j \in A^*$, $1 \leq j \leq p$; the compartment C_{t_i} will be replaced by p number of compartments; the j -th compartment, $1 \leq j \leq p$, of type t_{i_j} contains the same

objects as C_{t_i} , except x , which will be replaced by y_j ; all the links of C_{t_i} are inherited by each of the newly created compartments.

- **(b2) membrane dissolution rule:** $\square_{t_i} \rightarrow \lambda \{g\}$; the compartment C_{t_i} will be destroyed together with its objects and the links.
- **(b3) link creation rule:** $[x]_{t_i}; \square_{t_j} \rightarrow [y]_{t_i} - \square_{t_j} \{g\}$; the current compartment is linked to a compartment of type t_j and x is transformed into y ; if more than one compartment of type t_j exist and they are not linked with C_{t_i} , then one of them will non-deterministically be selected.
- **link destruction rule:** $[x]_{t_i} - \square_{t_j} \rightarrow [y]_{t_i}; \square_{t_j} \{g\}$; is the link between compartments are removed that is the compartments are disconnected.

The kP system rules represent molecular interaction or chemical reactions inside compartments. If the products of a reaction remain inside the same compartment, then the reaction is expressed with the rewriting rules, but if a product needs to be transferred to another membrane, then communication rules should be used. The rewriting and communication rules do not change the structure and the number of the compartments. The membrane division rules can model the proliferation of the biological systems, e.g. cells, whereas the membrane dissolution models the termination of membrane functionality or cell death. The link creation and destruction rules enable dynamically changing the connections between the compartments, which may happen in living cells when a cell changes its location.

2.2.4 kP systems execution strategies

Execution strategies define how the rules will be executed for each compartment type t from T —see Definition 1.

Definition 6 For a compartment type $t = (R, \sigma)$ from T and $r \in \text{Lab}(R)$, $r_1, \dots, r_s \in \text{Lab}(R)$, the execution strategy, σ , is defined as follows:

- $\sigma = \lambda$ means no rule from the current compartment will be executed.

- $\sigma = \{r\}$ means the rule r is executed.
- $\sigma = \{r_1, \dots, r_s\}$ means one of the rules labelled r_1, \dots, r_s will non-deterministically be chosen and executed; if no rule is applicable then nothing is executed. This execution strategy is called *alternative execution strategy* or *choice execution strategy*.
- $\sigma = \{r_1, \dots, r_s\}^*$ means an arbitrary number of time the rules will non-deterministically be chosen and executed. This is called *arbitrary execution strategy*.
- $\sigma = \{r_1, \dots, r_s\} >$ all applicable rules will be executed until no applicable rule remains. This is the *maximal parallelism execution strategy*.
- $\sigma = \sigma_1 \& \dots \& \sigma_s$, applies the execution strategies sequentially $\sigma_1, \dots, \sigma_s$, where σ_i , $1 \leq i \leq s$, describes any of the aforementioned execution strategies; if one of σ_i fails to be executed then the remained $\sigma_i \dots \sigma_s$ will no longer be executed. This is called *sequential execution strategy*.
- for any execution strategy σ , only one single structure changing rule is allowed.

Choice execution strategy non-deterministically selects and applies only one rule among the applicable rules. Arbitrary execution strategy, at each step, an arbitrary number of times non-deterministically chooses and applies the applicable rules. Maximal parallelism execution strategy applies all rules at each step, the order of applying the rules is not important. Sequence execution strategy applies all rules sequentially, namely, the rules are applied in the same order as they are defined.

A *configuration* of a kP system with n compartments, C_1, \dots, C_n , is a tuple $c = (u_1, \dots, u_n)$, where u_i is a multiset of compartment C_i , $1 \leq i \leq n$. Structure changing rules might be executed which may change the compartment number. A *configuration* $c' = (v_1, \dots, v_m)$ follows in one step from $c = (u_1, \dots, u_n)$, if in each C_i the σ_i is applied to u_i . A *computation* is a finite sequence of steps starting from the initial configuration, (w_1, \dots, w_n) , and at each step applying the rules of the execution strategies of each compartment.

2.3 kP-Lingua

kP-Lingua is the modelling language of kP systems which can express the kP systems specifications in a machine-readable format. kP-Lingua expresses kP system models in an unambiguous, concise and intuitive manner [41, 52, 70, 71]. kP-Lingua can express non-deterministic systems, but it does not support stochastic systems.

Example 2 illustrates how a kP system can be modelled with the kP-Lingua. The example is remodelled by adding new rules, execution strategies, and a new compartment instance to the example in Section 5.

C1 and C2 are two compartment types in Example 2. C1 has one compartment instance, m1, and C2 has two instances m2 and m3. m1 starts with a multiset of two a objects and one b. One a and five a are the initial multisets of m2 and m3, respectively. m1 is linked to m2 and m3, but there is no link between m2 and m3. The C1 compartment type has four different execution strategies. The scopes of the execution strategies are determined with the curly brackets, except the sequence strategy, which is the last strategy, and it does not have a reserved keyword.

The first execution strategy of C1 is a choice strategy which non-deterministically selects and applies only one rule at each step. The first rule has a guard, to apply the rule the guard should be true; namely, there should be more than two b objects in the compartment. The first rule also a communication rule, if the rule is applied, then one b and one a objects will be produced and the object b will remain in the same compartment, but the object a will be sent to an instance of C2 compartment type. As there are two instances of C2 (m2 and m3) are connected to m1, one of them will randomly be selected as the target for transmitting the object. The second rule of the choice strategy is a rewriting rule; it indicates that if the rule is applied one b object will be consumed and two b objects will be produced. The objects will remain in the same compartment.

The second execution strategy in the compartment type C1 is an arbitrary execution strategy, the rules in this scope are a random number of times will non-deterministically be selected and applied. If the first rule of the arbitrary execution strategy is applied, then

one b will be consumed, and nothing will be generated. It represents the degradation of the molecules. The other two rules inside arbitrary execution strategy are also rewriting rules.

The third strategy is a maximum parallelism which is specified by ‘max’ keyword. The strategy contains the same rules of the arbitrary strategy. The rules inside this block are exhaustively applied until no applicable rule remains.

The last execution strategy inside C1 compartment type is a sequential execution strategy. The sequential execution strategy applies rules in the same order as they are defined. If one of the rules is not applicable, then the rest of the rules will not be applied. The first rule of the sequential execution strategy is a simple rewriting rule. The second rule is a dissolution rule which is a structure changing rule associated with a guard. If a compartment contains exactly three a objects and at least two c objects, then the rule will be applicable, and hence, the compartment will dissolve, and it will not function anymore.

The compartment type C2 has only one choice execution strategy which has two communication rules. If the first rule is applied, the compartment will try to transmit two c objects to a C2 type compartment. However, in the example, there is no link between m2 and m3; therefore, even after the rule is executed one a will be consumed, but the c objects cannot be transmitted to any target compartment instance. The second rule is the same as the first one, except it will send the c objects to an instance of C1 compartment type. Therefore, both m2 and m3 compartments will send their c objects to m1.

As a case study, a biomolecular system example modelled in kP-Lingua is demonstrated in Section 2.5.1. Additionally, the description of the kP-lingua language with a synthetic biology example is provided in Section 5. For further kP system models, please see the kPWorkbench website [72] where you can find various case studies, and you can download their models described in kP-Lingua. The grammar of kP-Lingua in Extended Backus-Naur Form (EBNF) form provided in [52].

Example 2 *Modelling a kP system using kP-Lingua.*

```
1      //compartment type C1
2      type C1 {
3          choice {
4              > 2b : 2b -> b, a(C2).
5              b -> 2b.
6          }
7          arbitrary {
8              b -> {}.
9              a, b -> c.
10             c -> a, b.
11         }
12         max {
13             b -> {}.
14             a, b -> c.
15             c -> a, b.
16         }
17         a-> b.
18         =3a: 2c ->#.
19     }
20     //compartment type C2
21     type C2 {
22         choice {
23             a -> {2c}(C2).
24             a -> {2c}(C1).
25         }
26     }
27     //compartment instantiation
28     m1 {2a, b} (C1).
29     m2 {a} (C2).
30     m3 {5a} (C2).
31     //linking compartments
32     m1 - m2.
33     m1 - m3.
```

2.4 Stochastic P systems

Stochastic P systems (SP systems) are the probabilistic variant of P systems. As we stated previously, P systems apply rules in a maximally parallel way. However, the maximal parallelism execution strategy has drawbacks. For example, all rules have equal time steps

which do not represent the real system reactions [19, 55]. Therefore, the stochastic variant of P systems has been devised, which associates stochastic constants to rules. The stochastic constants are used for calculating the probability of a rule to be chosen and applied, and it is also used for measuring the required time for applying the rules [9, 19, 55]. Although the first paper in Chapter 5 has the formal definition of Stochastic P systems, for the sake of completeness and consistency, we added it here too. Therefore, whenever possible, we used the same symbols that are used for defining the preliminaries in 2.2.1 and the kP systems in 2.2.

Definition 7 *A stochastic P system (SP system) with a single compartment is a tuple:*

$$SP = (A, w, R) \quad (2.2)$$

where A is a finite set of objects, i.e. alphabet; w is the finite initial multiset of objects of the compartment, an element of A^* ; R is a set of multiset rewriting rules, of the form $r_k : x \xrightarrow{c_k} y$, where x, y are multisets of objects, $x \in A^+$ and $y \in A^*$ (y might be empty).

A finite set of labels is L , and a population of SP systems indexed by this family is SP_h , $h \in L$. A lattice, denoted by Lat , is a bi-dimensional finite array of coordinates, (a, b) , with a and b positive integer numbers. Now we can define a lattice population P system.

Definition 8 *A lattice population P system (LPP system) is a tuple*

$$LPP = (Lat, (SP_h)_{h \in L}, Pos, Tr) \quad (2.3)$$

where Lat , SP_h and L are as above and $Pos : Lat \rightarrow \{SP_h | h \in L\}$ is a function associating to each coordinate of Lat a certain SP system from the given population of SP systems. Tr is a set of translocation rules of the form $r_k : [x]_{h_1} \xrightarrow{c_k} [x]_{h_2}$, where $h_1, h_2 \in L$; this means that the multiset x from the SP system SP_{h_1} , at a certain position in Lat , will move to Lat that contains an SP system SP_{h_2} .

c_k that appears in both definitions above is the stochastic constant which is used for computing the next rule to be applied in the system.

The lattice population system can be seen as the tissue structures in the biological systems, where an SP system can represent a cell. The compartments can interchange molecules only with their neighbours, namely east, west, south and north neighbours.

In Section 5 a synthetic biology example modelled in stochastic P system is provided in greater details.

2.5 Case Study - Gene expression

In this section, we want to concretise the computational model approaches with a biomolecular system example, which has been previously studied in [19]. The example represents constitutive, down-regulation and up-regulation of gene expression. The gene expression system starts with a gene and optionally with transcriptional factors (either activator or repressor which are DNA or RNA binding proteins). The gene produces (transcripts) the messenger RNA (mRNA) molecule. The activator molecules can promote the transcription process or the repressor molecules can block it. Finally, mRNA translation produces the target protein. For the sake of simplicity, we assume the whole process occurs inside one membrane.

We will model this example with two classes of P systems, namely kernel P systems, and stochastic P systems, and also in the following section, we will use this example for explaining two other formalisms, namely the π -calculus and Petri-nets. We have designed the kP system model of the example from scratch. The stochastic P systems model in Table 2.3 and π -calculus models in Table 2.3 are taken from [19]. We have also adapted the Petri-nets model in Figure 2.2 from [19].

2.5.1 kP Systems (kP-Lingua)

The model in [19] associates constants with rules, which are used for calculating the probability of a rule to be applied, and the time required for applying the rules [19]. However, kP systems only support deterministic and non-deterministic systems, and do not support stochastic systems. Therefore, we ignore these constants to model the non-deterministic variant of the system. Table 2.2 represents the kP System of the gene expression example which is described with the kP-Lingua modelling language.

kP-Lingua requires specification of compartment types. Therefore we introduced compartment *typeB* which is specified at the first line of the model. The compartment type is instantiated with the compartment *b*, in the last line. The initial content of the compartment consists of a *gene* molecule and multisets of the transcriptional factors, namely 10 *activator* (*act*) and 10 *repressor* (*rep*) molecules. The rewriting rules within the maximal parallelism block, which is specified with the *max* keyword, are applied in the non-deterministic and maximally parallel way. That means that at each time step all applicable rules are non-deterministically chosen and exhaustively applied until there is no applicable rule remaining. The first and the second rewriting rules represent the transcription and translation processes, respectively. Rules *r3* and *r4* represent molecule degradation where an *RNA* or a *protein* molecule is consumed, but nothing produced. The rules *r5*, *r6* are reverse of each other and in *r7* *act_gene* produces an *RNA* but it does not change. Therefore, *r5*, *r6* and *r7* express that the activator molecules (*act*) can increase protein translation. Whereas *r8* and *r9* express how the repressor molecules (*rep*) postpone the transcription of the *gene* molecule into *RNA*. The kP-Lingua model is accessible from www.github.com/meminbakir/kernelP-store.

2.5.2 Stochastic P Systems

Similar to the kP systems, the stochastic P systems represent the initial condition of the system and the molecules with multiset of objects. The rewriting rules represent the transcription and the translation processes, but now the rules have constants that are used

```

type B { //compartment type
  max { //execution strategy
    // Rewriting rules
    /* r1 */ gene → gene, RNA .
    /* r2 */ RNA → RNA, protein .
    /* r3 */ RNA → {} .
    /* r4 */ protein → {} .
    /* r5 */ act, gene → act_gene .
    /* r6 */ act_gene → act, gene .
    /* r7 */ act_gene → act_gene, RNA .
    /* r8 */ rep, gene → rep_gene .
    /* r9 */ rep_gene → rep, gene .
  }
}
b {gene, 10act, 10rep}(B).

```

```

/* b is the compartment, {gene, 10act, 10rep} is its initial content, and B is its type,
namely compartment type */

```

Table 2.2: kP-Lingua model of the gene expression.

for determining the probability and the time required for applying the rules according to Gillespie algorithm [19, 56]. Table 2.3 shows the stochastic P system model of the gene expression system [19]. Π represents the stochastic P System model. The system has a single compartment, labelled b , and can be in three distinct initial multisets of objects, which are denoted by $M_{0,1}$, $M_{0,2}$, $M_{0,3}$ which has the *gene* molecule alone or together with the transcriptional factors, *10act* or *10rep*. The constitutive gene expression is represented by $r1 - r4$ rules, which simply consists of the transcription and the translation processes, and the degradation of *RNA* and *protein* molecules. The rules $r5 - r7$ represent that the activator molecules promote the translation, whereas the rules $r8$ and $r9$ inhibit the transcription, and hence, the translation process.

2.6 Other formalisms

In the previous sections, we introduced two variants of P systems. kP systems and its

$\Pi = (\{\text{gene, RNA, protein, act, act_gene, rep, rep_gene}\}, b, []_b, (b, M_i), r_1, \dots, r_9)$
$M_{0,1} = \text{gene}$
$M_{0,2} = \text{gene} + 10\text{act}$
$M_{0,3} = \text{gene} + 10\text{rep}$
Rules:
$r1 : [\text{gene}]_b \xrightarrow{c1} [\text{gene} + \text{RNA}]_b \quad c1 = 0.347 \text{ min}^{-1}$
$r2 : [\text{RNA}]_b \xrightarrow{c2} [\text{RNA} + \text{protein}]_b \quad c2 = 0.174 \text{ min}^{-1}$
$r3 : [\text{RNA}]_b \xrightarrow{c3} []_b \quad c3 = 0.347 \text{ min}^{-1}$
$r4 : [\text{protein}]_b \xrightarrow{c4} []_b \quad c4 = 0.00116 \text{ min}^{-1}$
$r5 : [\text{act} + \text{gene}]_b \xrightarrow{c5} [\text{act_gene}]_b \quad c5 = 6.6412087 \text{ molec}^{-1}\text{min}^{-1}$
$r6 : [\text{act_gene}]_b \xrightarrow{c6} [\text{act} + \text{gene}]_b \quad c6 = 0.6 \text{ s}^{-1}$
$r7 : [\text{act_gene}]_b \xrightarrow{c7} [\text{act_gene} + \text{RNA}]_b \quad c7 = 3.47 \text{ min}^{-1}$
$r8 : [\text{rep} + \text{gene}]_b \xrightarrow{c8} [\text{rep_gene}]_b \quad c8 = 6.6412087 \text{ molec}^{-1}\text{min}^{-1}$
$r9 : [\text{rep_gene}]_b \xrightarrow{c9} [\text{rep_gene} + \text{gene}]_b \quad c9 = 0.6 \text{ min}^{-1}$

Table 2.3: Stochastic P model of the gene expression, taken from [19]

modelling language kP-Lingua are particularly relevant to this thesis. However, there are many other formalisms established which can be used for describing the biological systems. In the following sections, we will briefly outline some of these formalisms and some of the specification languages which are used for specifying and designing biological systems.

2.6.1 Petri nets

Petri nets are a mathematical and computational modelling language which form directed graphs. A Petri net consists of two types of nodes, *places* are signified by circles and *transitions* are signified by bars (or boxes) [98]. The edges which connect transitions to places or vice versa are called *arcs*, and each arch is associated with weights. If an arc directs from a place to a transition, the place is called *input place*, and if it directs from a transition to a place, the place is called *output place*. Places may contain *token* which creates a configuration of the system, called *marking*. A transition can be *enabled* if the input places have enough tokens (not less than the weight of its outgoing arch), after a transition *fires* the required number of tokens will be consumed from input places, and

Biological System	Petri nets
Molecule	Place
Molecular population	Marking
Biochemical transformation	Transition
Reactant	Input place
Product	Output place

Table 2.4: Mapping between biological systems and Petri nets [19].

tokens will be produced in output places. If a place is the input of multiple transitions, i.e. multiple transitions are enabled, then one of the transitions will be selected and fired non-deterministically. If a transition is enabled and it has more than one output place, then all output places will get the corresponding number of tokens, which represents parallel execution.

Petri nets are suitable formalism for modelling unpredictable (non-deterministic, and stochastic), and concurrent biological systems [46, 61]. A Petri net model assigns each molecular species of a biological system to a place and the population of molecules at a specific time are represented by markings [19, 46, 61]. Transitions represent the biochemical transformations and reactions. Table 2.4 describes mapping between biological systems and Petri nets.

Figure 2.2 demonstrates the Petri net model of the gene expression example introduced in Section 2.5. Each molecule is represented by a place, and the transformation rules are represented by transitions r_i , $1 \leq i \leq 9$. Each arch has a weight which signifies the required number of tokens to enable the transition. Constitutive gene expression is set as the initial marking where only the *gene* place has one token. If the place *act* has $1 \leq n \leq 10$ then the system will have positive regulation, whereas if the place *rep* has $1 \leq n \leq 10$ then the system will have negative regulation [19]. The model is designed using the Platform

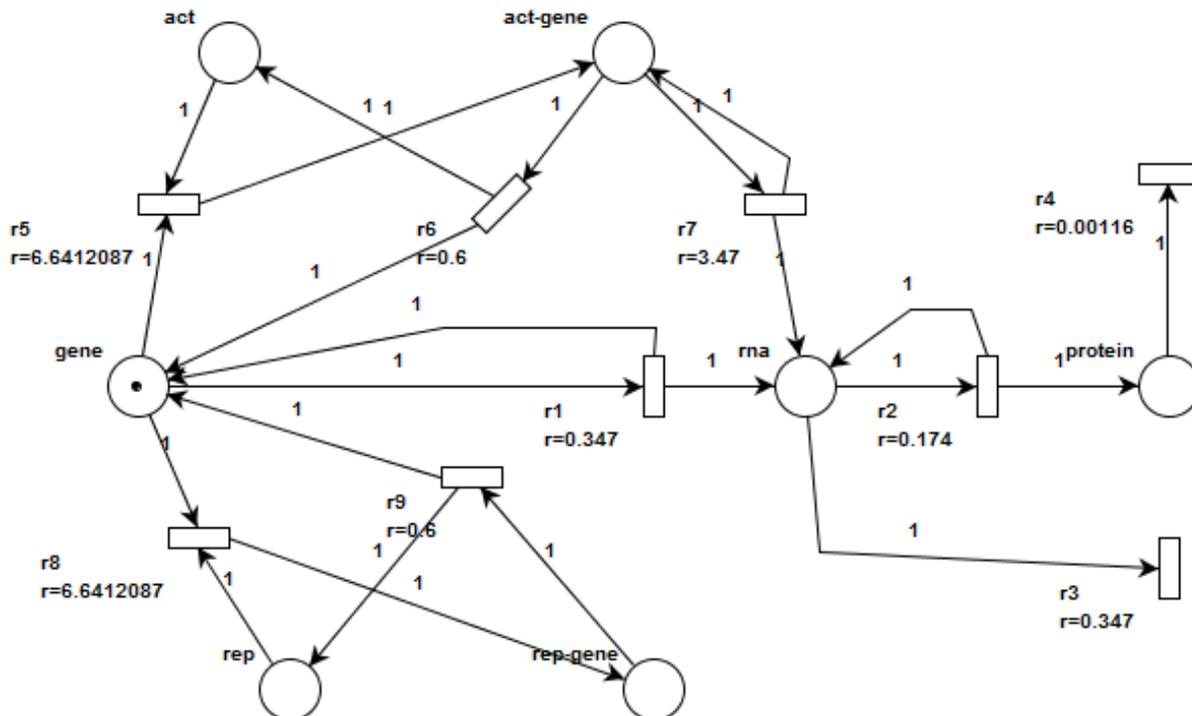


Figure 2.2: Stochastic Petri net model of gene expression (adapted from [19])

Independent Petri Net Editor 2 (PIPE2) tool [92].

2.6.2 π -calculus

π -calculus (or pi-calculus) is a member of process calculi which is designed for modelling concurrent system. π -calculus provides a high-level description for introducing parallel processes, new channels, communication between processes via channels, replication of processes, and non-determinism [77, 94, 105]. The π -calculus formalism is a good candidate for describing biological systems because the nature of biological systems is massively concurrent, and we can define a one-to-one mapping between two systems. To represent biological systems behaviour; each molecule is assigned to a process, and the quantity of molecules can be imitated by running replication of process in parallel. Molecule interaction is modelled with communication between processes via channels [19, 46]. Table 2.5

Biological System	π -calculus
Compartment	Private communication channel
Molecule	Process
Molecular population	Systems of communicating processes
Biochemical transformation	Communication channel
Compartment translocation	Private channel scope extrusion

Table 2.5: Mapping between biological systems and π -calculus [19]

represents mapping between biological systems and π -calculus. π -calculus allows developing models incrementally and composing them to build larger models. For more details on π -calculus and different extensions of π -calculus, such as stochastic π -calculus [93], κ -calculus, which have been used for modelling wide range of biological phenomena, please refer to [38, 46, 95, 96].

Figure 2.3 shows the π -calculus representation of the gene expression example from Section 2.5. This model is taken from [19]. Each process definition represents a molecule and its interactions with other molecules. The first order reactions, transformation and degradation, are assigned with a stochastic delay τ_k [19, 99]. For example, the gene process has been associated with τ_{c1} , after that time delay gene turns into gene and RNA. Similarly, the protein process corresponds degradation mechanism by becoming an inert process after τ_{c4} time delay. The higher order reactions have complementary communication channels, e.g., $a_{c5}?$, $a_{c5}!$, which processes represent the reactant molecules. The processes and reactants interact with the $c5$ constant.

2.6.3 Mathematical models

Until recently, the vast majority of biological systems have been modelled on the mathematical theory of differential equations, mostly using ordinary differential equations (ODEs) though sometimes using partial differential equations (PDEs) [11, 19, 25, 99]. Differential

$S_{0,1} = \text{gene}$
 $S_{0,2} = \text{gene} + \text{act} + \dots + \text{act}$
 $S_{0,3} = \text{gene} + \text{rep} + \dots + \text{rep}$

Component processes
 $\text{gene} := \tau_{c1}. (\text{gene} \mid \text{RNA}) + a_{c5}?. \text{act_gene} + r_{c8}?. \text{rep_gene}$
 $\text{RNA} := \tau_{c2}. (\text{RNA} \mid \text{protein}) + \tau_{c3}. 0$
 $\text{protein} := \tau_{c4}. 0$
 $\text{act} := a_{c5}!.0$
 $\text{act_gene} := \tau_{c6}. (\text{act} \mid \text{gene}) + \tau_{c7}. (\text{act_gene} \mid \text{RNA})$
 $\text{rep} := r_{c8}!.0$
 $\text{rep_gene} := \tau_{c9}. (\text{rep} \mid \text{gene})$

Figure 2.3: π -calculus model of gene expression (taken from [19]).

equations are well suited for modelling molecular interactions (or chemical reactions) in cellular systems [46]. ODE defines each model species as a variable which represent the concentration (population) over time [25]. The realisation of the models relies on two assumptions [19, 25, 99]: The first assumption is that the concentration of the molecules does not change with respect to space, that is, the number of each molecule is large enough, and they are well mixed. The second assumption is that the concentration changes smoothly over time.

When the number of reacting species is low, the reactions are slow and separated by irregular intervals, which is a general case in biological systems, these assumptions are no longer valid, and this approach becomes problematic [25, 99]. In such a situation, the reactions do not take place continuously which means the system represents discrete and stochastic behaviour [25, 99]. The Chemical Master Equation (CME) is a system of many coupled ODEs that describes the evolution chemical reactions as a stochastic system (in a well-mixed, fixed temperature and fixed volume) [111, 117]. Therefore, The ODEs based on CME can capture the stochastic behaviour of the reacting system which is a more usual case in real systems. However, as CME is a formulation of many ODEs they can be solved efficiently only for a few simple systems [25].

2.6.4 Specification languages

In the previous sections, we summarised some of the computational modelling formalism for describing the biological systems. In addition to computational models, there are some attempts to describe and design biological systems with high-level languages, especially with the growth of synthetic biology these languages are more utilised [88].

GenoCAD [30, 49] is the first computer-assisted-design (CAD) application for synthetic biology, built upon the foundation of the theory of formal languages [17, 30]. DNA segments are called parts, such as promoters, transcription terminators, genes, protein domains, which are defined as the terminals of the language [116]. Design strategies have rules to specify which parts and classes of parts, called categories (non-terminals), can be combined. The design strategies also make sure that the categories can be used only in correct order [116]. GenoCAD is implemented as a web-based tool. GenoCAD library includes numerous distinct basic parts. Also, it allows users to create or import their private parts without having to share them with other users [49, 116]. The web application guides the users to form constructs through a comfortable “point and click” fashion. The application recently has been extended to allow designing BioBrick [22] constructs compliant with different assembly standard [17, 31].

Eugene is a human and machine readable language developed for designing biological DNA sequences, parts, devices (composite of multiple fundamental parts), and Systems [17, 21, 45]. Eugene is inspired by the approach used by the Electronic Design Automation (EDA) industry (e.g. Verilog, VHDL); it provides a biological design netlist (a collection of abstract components and their connections) which can be transformed into corresponding physical implementations [21]. The essential features of Eugene include: (i) Specifying synthetic biological components at the various level of abstraction, e.g., properties, part instance, parts, devices. (ii) Defining constraints and rules on the component composition, the constraint system allows the automatic generation of composite devices from a set of separate parts. (iii) Eugene can directly interact with some external design applications, which extract data from repositories of biological parts and encapsulate that

data as Eugene “header files”. The header files provide modularity feature to Eugene. Eugene ultimately produces collections of devices which are composed of specific constraints, and parts and properties. Eugene tool does not have a built-in simulation tool, but it translates the custom Eugene data structures to an exchange format for external simulators [21].

Genetic Engineering of Cells (GEC) is another high-level language that aims to facilitate the design, analysis, and implementation of biological devices. GEC utilises previous research in the synthetic biology field, includes integrating MIT Registry of Standard Biological Parts (<http://parts.igem.org>) notions with experimental techniques to combine these parts into higher-level devices [86, 87]. GEC enable users to design devices with basic knowledge of the available part (elementary) types, namely promoters, ribosome bindings sites, protein coding regions and terminators [17, 86, 87]. The elementary parts (devices) can be combined, and the properties of the parts can be expressed as constraints [17, 86, 87]. The tool based on GEC is called as Visual GEC, which provides designing and simulation functionalities of biological parts [87].

In this section, we summarised three widely-used high-level languages for specifying biological part and devices. But there are some others such as, Kera [114], Antimony [110], Proto [16]. They are beyond the scope of this thesis. However, interested reader can refer to [17] which summarises some of these languages.

Chapter 3

Analysis

After specifying a model, usually, the next objective is to analyse it. Thanks to the executable feature of computational models, we can test hypotheses and analyse interesting model properties [11]. This research mainly investigated the model checking techniques for analysis purpose. We cover the methods and tools used for model checking in the following sections and the Part II that subsume our published studies.

3.1 Model checking

Model checking is a powerful verification technique, which has been widely used in computer systems, as well as for biological system models [41]. It formally demonstrates the correctness of desired properties of a model by exhaustively investigating its entire state space, namely, considering all execution paths. Therefore, the approach guarantees the accuracy of the properties. This approach is called *exact* model checking and the model checking tools that implement this feature is called exact model checkers.

Model checking comprises three basic steps [7, 11, 64].

Modelling:

The system is expressed in a high-level modelling language accepted by the model checking tool, which provides an unambiguous representation of the input system.

Specification:

The desired properties are specified with temporal logic formulas which enable querying model behaviour over time. This is covered in the next section.

Verification:

The validity of each property is verified on the model. For a model \mathcal{M} and a property ϕ , the model checking algorithm decides if ϕ is valid on \mathcal{M} , which can be denoted as $\mathcal{M} \models \phi$. For non-probabilistic models, the verification output is, typically, either ‘true’ or ‘false’. If the property is false, then a counter-example may be produced. For probabilistic systems, the output can be some estimate of the ‘probability of correctness’ [11].

Model checking was initially implemented for analysing transition systems. A *transition system* regards time as discrete, and describes a set of states and the possible transitions between them, where each state represents some instantaneous configuration of the system. Formally, a transition system is defined as follows [64].

Definition 9 *A transition system is $\mathcal{M} = (S, \rightarrow, L)$ where S is the set of states, and \rightarrow represents the binary transition relation, such that every $s \in S$ goes to some $s' \in S$ with $s \rightarrow s'$, and a labelling function $L : S \rightarrow \mathcal{P}(\text{Atoms})$ where $\mathcal{P}(\text{Atoms})$ denotes the power set of atomic propositions.*

More recently, model checking has been extended by assigning probabilities to state transitions (*probabilistic model checking*). In the probabilistic version of transition systems, the transition from the current state to target states are determined by some probabilities, rather than deterministic or non-deterministic choices. Probabilistic models are useful for verifying quantitative features of systems. *Markov chains* (MCs) are typical models for presenting such systems [7]. Markov chains are transition systems where each state has a probability distribution that specifies the transition probabilities from the current state to its successors. The probability distribution from the current state to the next state only depends on the current state, namely not on the history of how the system got

there from the initial state[7]. This distinguishing feature of MCs is known as memoryless property[7]. In practice, the probabilistic model checking commonly implements the discrete-time Markov chains (DTMC), and continuous-time Markov chains (CTMC) for representing probabilistic systems. Now we shall formally introduce them first, and then we can move to temporal logics to show how they are verified.

Definition 10 *A labelled discrete-time markov chain (DTMC) is;*

a tuple $\mathcal{M} = (S, P, s_i, AP, L)$ where S is a finite set of states, $s_i \in S$ is the initial state, $P : S \times S \rightarrow [0, 1]$ is the transition probability function such that $\forall s, \sum_{s'_i \in S} P(s, s') = 1$. AP is a set of atomic proposition and L is a labelling function such that $L : S \rightarrow 2^{AP}$, assigns true atomic propositions to the related states.

Similar to the DTMC, the CTMC also have discrete state space, but the transition time between states of a CTMC system occurs in real-time, namely continuously (as its name indicates), whereas in DTMC it corresponds to a discrete time step [73]. CTMCs are also memoryless, namely the probability of moving from the current state to the next state depends only on the current state. Let's first introduce the formal definition of a CTMC [6, 73, 108].

Definition 11 *A labelled Continuous Time Markov Chain (CTMC) is a tuple $\mathcal{M} = (S, s_i, R, AP, L)$ where S is finite set of states, $s_i \in S$ is the initial state, $R : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is transition rate matrix. AP is a set of atomic proposition and L is a labelling function such that $L : S \rightarrow 2^{AP}$ which assigns the set of atomic propositions $L(s)$ that true in each state $s \in S$.*

Different from a DTMC model, each pair of states in the CTMC are assigned with a rate $R(s, s')$, and a transition between each pair of states s, s' can occur only if $R(s, s') > 0$. If $R(s, s') = 0$ transition between s and s' is not possible. If s has only one successor, s' , and $R(s, s') > 0$, then the probability of a transition occurring between s and s' within t time-units equals $1 - e^{-R(s, s').t}$. However, generally in CTMCs there are more than one

successors of the state s , which is known as *race condition*. To identify the next state of s in a race condition case, we need to determine the probability of transition to each successor of s , i.e. s' . The total transition rates from s is, $E(s)$ called as the *exit rate* s which is:

$$E(s) = \sum_{s' \in S} R(s, s') \quad (3.1)$$

If a state s has no outgoing transitions it is called *absorbing*, that is $E(s) = 0$. The probability of moving from s to s' within t time units, when s is not absorbing, is denoted as $P(s, s', t)$ where:

$$\mathbf{P}(s, s', t) = \frac{\mathbf{R}(s, s')}{E(s)} \cdot (1 - e^{-E(s) \cdot t}). \quad (3.2)$$

Basically, by the probability $P(s, s', t)$ the winner successor state of s is chosen [6]. Furthermore, the probability of transition from s to s' , $P(s, s')$, can be specified by converting the CTMC to a DTMC, whereby ignoring the time spent at any state, more details on this conversion can be found at [6, 73].

Before starting the temporal logics section which is used for specifying model properties, it might worth mentioning that we have extensively used different model checking approaches and tools during this study, some of the findings have been published and reported in this thesis. For less formal but more practical usage of model checkers, we refer readers to Chapter 5, 6, and 7.

3.2 Temporal logics

Model checking uses **temporal logics** as property specification language. Temporal logics are an extension of propositional logics. The difference is that temporal logics formulas can be true in some states and may be false in others which means that the truth of formulas may change from state to state. Regarding the structure of time, temporal logics can be either linear time or branching-time. Linear-time represents the time as a set of paths, for each time instant, there is a single successor. Hence, a path is a sequence of time instances.

Branching-time represents the time as a tree-like structure, where the root is the current time, and the branches go to future [7, 64].

In order to demonstrate each temporal logic formula, we will verify some properties of the gene expression model, described in Section 2.5. In the temporal logic formulas, we will refer the number of molecules with their name, for example, instead of “the number of protein molecules”, we will use “protein” only, therefore, $protein > 5$ is a propositional atom, because it means that the number of protein molecules is greater than 5.

3.2.1 Linear-Time Temporal Logic (LTL)

Linear-Time Temporal Logic (LTL) is used for querying transition systems (see Definition 9). LTL models the possible future of a system as a sequence of states, which is called path or computational path.

Definition 12 *A path of model a transition system \mathcal{M} (see Definition 9) is an infinite sequence of states $s_1 \rightarrow s_2 \rightarrow \dots$, $s_i \in S$ where, for $i \in \mathbb{N}^+$, $s_i \rightarrow s_{i+1}$. A path is specified as $s_1 \rightarrow s_2 \rightarrow \dots$*

Let $\pi = s_1 \rightarrow s_2 \rightarrow \dots$ be a path, and π^i means the starting state of the path is s_i . Now, we can introduce the syntax of LTL, the BNF form LTL is as follows [7, 64]:

$$\begin{aligned} \phi ::= & \top \mid \perp \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \\ & (\phi \rightarrow \phi) \mid (X\phi) \mid (F\phi) \mid (G\phi) \mid (\phi U \phi) \mid (\phi W \phi) \mid (\phi R \phi) \end{aligned}$$

where p is any propositional atom, and ϕ , ϕ_1 , and ϕ_2 are LTL formulas. The G , F , X , U , R , and W operators are called *temporal connectives*. G stands for “Globally”, which includes current state and all future states of a path, and $G\phi$ means that all states of a path satisfy ϕ . F means “eventually”, and $F\phi$ states that some states in future will satisfy ϕ . X stands for “Next”, and $X\phi$ means that the second state of a path satisfies ϕ . U stands for “Until”. Until is a binary operator and $\phi_1 U \phi_2$ means that ϕ_1 holds and it will keep holding until ϕ_2 holds. That is, ϕ_2 must eventually hold. R means “Release”, $\phi_1 R \phi_2$ states that ϕ_2 holds until ϕ_1 becomes true, namely ϕ_2 releases ϕ_1 . W stands for “Weak-until”, $\phi_1 W \phi_2$ it is like

Until operator but it does not require ϕ_2 eventually hold. [7, 18, 44, 64]. In the following section we provide the formal semantics of the LTL formulas.

Semantics of LTL Assume $\mathcal{M} = (S, \rightarrow, L)$ is a transition system and $\pi = s_1 \rightarrow s_2 \rightarrow \dots$ is a path, where π^i starts at state s_i , $i \in \mathbb{N}^+$. We define the satisfaction relation of LTL on π as follows [7, 18, 64]:

- $\pi \models \top$
- $\pi \not\models \perp$
- $\pi \models p$ iff $p \in L(s_1)$, where s_1 is the first state
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
- $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
- $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$
- $\pi \models X\phi$ iff $\pi^2 \models \phi$
- $\pi \models G\phi$ iff , for all $i \geq 1$, $\pi^i \models \phi$
- $\pi \models F\phi$ iff there is some $i \geq 1$ where $\pi^i \models \phi$
- $\pi \models \phi U \psi$ iff there is some $i \geq 1$ where $\pi^i \models \psi$ and for all $j = 1, \dots, i - 1$ we have $\pi^j \models \phi$
- $\pi \models \phi R \psi$ iff either there is some $i \geq 1$ where $\pi^i \models \phi$ and for all $j = 1, \dots, i$ we have $\pi^j \models \psi$, or for all $k \geq 1$ we have $\pi^k \models \psi$.
- $\pi \models \phi W \psi$ iff either there is some $i \geq 1$ where $\pi^i \models \psi$ and for all $j = 1, \dots, i - 1$ we have $\pi^j \models \phi$; or for all $k \geq 1$ we have $\pi^k \models \phi$

Example 3 *LTL examples for verifying the gene expression model (see Section 2.5).*

Property 1: “Eventually the protein molecule will be generated.”

LTL: $F(\text{protein} > 0)$

Property 2: “Protein will not be produced before an RNA molecule is produced.”

LTL: $\neg(\text{protein} > 0) U \text{RNA} > 0$

Example 3 demonstrates how to model check the gene expression model in Section 2.5 with LTL formulas the informal description of the properties are also provided. The LTL formulas are specified based on NuSMV language specification (please see [79] for the details of NuSMV language). For the first property a model checker will check all paths, for every path if there is a future state where the number of proteins is greater than zero, then the property will hold for the model, does not otherwise. For the second property to hold, again, the model checker expects for all paths the number of proteins is zero and remains zero until an RNA molecule is produced.

3.2.2 Computational Tree Logic (CTL)

Computational Tree Logic (CTL) is also used for querying the transition systems, and it considers time as branching. We define CTL in BNF as follows [64]:

$$\begin{aligned} \phi ::= & \top \mid \perp \mid p \mid (\neg\phi) \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2) \mid \\ & (\phi_1 \rightarrow \phi_2) \mid AX\phi \mid EX\phi \mid AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \mid A[\phi_1 U \phi_2] \mid E[\phi_1 U \phi_2] \end{aligned}$$

CTL has G, F, U , and X of temporal operators, in addition, it has A , (**All**) and E (**Exists**) path quantifiers which means “all paths” and “exists a path” respectively. CTL operators cannot exist without being preceded by one of the path quantifiers [7, 18, 44, 64]. Informally, $AX\phi$ says: “for all paths the state after the initial state ϕ holds”, whereas $EX\phi$ says: “there exists a path such that the state after the initial state ϕ holds”. Similarly, as informal descriptions of the rest of the operators (the formal descriptions are provided in the following section) [7, 18, 64, 69]:

- $AG\phi$: for all paths the property ϕ globally holds.
- $EG\phi$: there exists a path such that ϕ holds globally along the path.
- $AF\phi$: for all paths there will be some future state where ϕ holds.
- $EF\phi$: there exists a path such that ϕ holds in some future state.
- $A[\phi_1 U \phi_2]$: all paths satisfy $\phi_1 U \phi_2$ on them.
- $E[\phi_1 U \phi_2]$: there exists a path such that $\phi_1 U \phi_2$ holds on it.

Semantics of CTL Assume $\mathcal{M} = (S, \rightarrow, L)$ is a model of transition systems and $s \in S$, ϕ is a CTL formula. For the paths in \mathcal{M} starting at s , $\mathcal{M}, s \models \phi$ is defined as following [7, 18, 64, 69]:

- $\mathcal{M}, s \models \top$ and $\mathcal{M}, s \not\models \perp$
- $\mathcal{M}, s \models p$ iff $p \in L(s)$
- $\mathcal{M}, s \models \neg\phi$ iff $\mathcal{M}, s \not\models \phi$
- $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \phi_1$ and $\mathcal{M}, s \models \phi_2$
- $\mathcal{M}, s \models \phi_1 \vee \phi_2$ iff $\mathcal{M}, s \models \phi_1$ or $\mathcal{M}, s \models \phi_2$
- $\mathcal{M}, s \models \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, s \not\models \phi_1$ or $\mathcal{M}, s \models \phi_2$
- $\mathcal{M}, s \models AX\phi$ iff $\forall s_1$, where $s \rightarrow s_1$ and $\mathcal{M}, s_1 \models \phi$.
- $\mathcal{M}, s \models EX\phi$ iff $\exists s_1$, where $s \rightarrow s_1$ and $\mathcal{M}, s_1 \models \phi$.
- $\mathcal{M}, s \models AG\phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , and all s_i along the path, where $i \in \mathbb{N}^+$ and $\mathcal{M}, s_i \models \phi$.
- $\mathcal{M}, s \models EG\phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , and all s_i along the path, $i \in \mathbb{N}^+$ and $\mathcal{M}, s_i \models \phi$.

Example 4 *CTL examples for verifying the gene expression model (see Section 2.5).*

Property 1: “There exists a path which eventually lead to the production of the protein molecule.”

CTL: $EF(\text{protein} > 0)$

Property 2: “For all paths protein will not be produced before an RNA molecule is produced.”

CTL: $A[\neg(\text{protein} > 0) U \text{RNA} > 0]$

- $\mathcal{M}, s \models AF\phi$ holds iff for all path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , there is some $s_i, i \in \mathbb{N}^+$, such that $\mathcal{M}, s_i \models \phi$.
- $\mathcal{M}, s \models EF\phi$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , and for some s_i along the path, $i \in \mathbb{N}^+$ and we have $\mathcal{M}, s_i \models \phi$.
- $\mathcal{M}, s \models A[\phi_1 U \phi_2]$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , that path satisfies $\phi_1 U \phi_2$, i.e. there is some s_i along the path, $i \in \mathbb{N}^+$, where $\mathcal{M}, s_i \models \phi_2$, and, for each $1 \leq j < i$, we have $\mathcal{M}, s_j \models \phi_1$.
- $\mathcal{M}, s \models E[\phi_1 U \phi_2]$ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$, where s_1 equals s , and that path satisfies $\phi_1 U \phi_2$ as specified previously.

Similar to the LTL example, Example 4 provides exemplary CTL formulas for verification of the gene expression example in Section 2.5, with informal description of the properties. Since the first property has Exists (E) path quantifier, the model checker starts from a given state and checks all paths, if there exists a path where the number of proteins eventually becomes greater than zero, then the property will hold. The path quantifier of the second property is All (A); therefore it will be same as the second property of LTL, in Example 3.

As another example, the verification of a kP System model with CTL provided in Section 5.

3.2.3 Probabilistic Computation Tree Logic (PCTL)

Probabilistic temporal logics formulate conditions on the probabilistic version of transition systems where the transition from the current state to target states determined by some probabilities, rather than deterministic or non-deterministic choices. Probabilistic Computational Tree Logic (PCTL) is used for verifying Discrete Time Markov Chain (DTMC) models—see Definition 10.

Paths represent the executions of DTMC models. A *path* of a DTMC model is defined as infinite sequences, more formally; for a DTMC $\mathcal{M} = (S, P, s_i, AP, L)$, a path π is a sequence of s_1, s_2, \dots , such that $s_i \in S$ where $i \in \mathbb{N}^+$ and $P(s_i, s_{i+1}) > 0$. π^i means that the starting state of the path is s_i , and the set of paths of a DTMC model \mathcal{M} starting in state s is denoted $Path^{\mathcal{M}}(s)$ [7, 73].

Now we shall continue with the description and the formal definition of PCTL which is used for formulating conditions on DTMCs. PCTL is an extension of CTL temporal logic, the most prominent difference is that PCTL excludes the path quantifiers. Instead, it introduces probabilistic quantification with the probabilistic operator [7, 73]. The syntax of PCTL over state and path formulas in BNF form is like the following [7, 73]:

$$\begin{aligned} \phi & ::= true \mid a \mid (\phi_1 \wedge \phi_2) \mid \neg\phi \mid P_{\sim p}(\psi) \\ \psi & ::= X\phi \mid \phi_1 U \phi_2 \mid \phi_1 U^{\leq n} \phi_2 \end{aligned}$$

where a is an atomic proposition, ϕ , ϕ_1 , and ϕ_2 are state formulas, ψ is a path formula, $\sim \in \{<, >, \leq, \geq\}$, $p \in [0, 1]$ interval, and $n \in \mathbb{N}$. X (neXt), and U (Until) temporal operators have the same meaning as CTL operators, namely $X\phi$ is true if ϕ holds in the next state of the path and $\phi_1 U \phi_2$ is true if ϕ_2 holds in a step in the future and ϕ_1 is true up until that step [73]. Different from CTL, in PCTL they are always preceded by probabilistic operator, i.e. $P_{\sim p}[\cdot]$. $U^{\leq n}$ operator is bounded until operator, and the $\phi_1 U^{\leq n} \phi_2$ formula asserts that ϕ_1 is true and it will stay true until it reaches a ϕ_2 -true state within n states of the path.

Semantics of PCTL Given a DTMC model $\mathcal{M} = (S, P, s_i, AP, L)$, $s \in S$ and ϕ, ϕ_1, ϕ_2 are PCTL state formulas, and ψ be a PCTL path formula. The satisfaction relation \models is defined as [7, 73]:

- $s \models \text{true}$ for all $s \in S$
- $s \models a \iff a \in L(s)$
- $s \models \neg\phi \iff s \not\models \phi$
- $s \models \phi_1 \wedge \phi_2 \iff s \models \phi_1$ and $s \models \phi_2$
- $s \models P_{\sim p}[\psi] \iff \text{Prob}^M(s, \psi) \sim p$
 where $\text{Prob}^M(s, \psi) \stackrel{\text{def}}{=} Pr_s\{\pi \in \text{Path}^M(s) \mid \pi \models \psi\}$

Given any path $\pi \in \text{Path}^M(s)$:

- $\pi \models X\phi \iff \pi^2 \models \phi$
- $\pi \models \phi_1 U \phi_2 \iff \exists j \in \mathbb{N}^+$ such that $\pi^j \models \phi_2$ and $1 \leq i < j$ $\pi^i \models \phi_1$ for $i \in \mathbb{N}^+$
- $\pi \models \phi_1 U^{\leq n} \phi_2 \iff \exists j \in \mathbb{N}^+$ such that $j \leq n$ and $\pi^j \models \phi_2$, and $1 \leq i < j$ $\pi^i \models \phi_1$ for $i \in \mathbb{N}^+$

Remark 1 Before starting the PCTL examples, we want to note that one can easily derive false, \vee , \rightarrow logical operators from the \neg and \wedge operators. Additionally, F (Eventually) temporal operator, where $F\phi$ means that ϕ will eventually hold, and G (Globally) temporal operator, where $G\phi$ means ϕ holds in every state of the path, can be derived from U operator like the following [58, 73]:

- $\text{false} \equiv \neg \text{true}$
- $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
- $P_{\sim p}[F \psi] \equiv P_{\sim p}[\text{true} U \psi]$

Example 5 *PCTL examples for verifying the the gene expression model in Section 2.5.*

Property 1: “The probability that the system will eventually produce a protein molecule is greater than 0”

$$\text{PCTL: } P_{>0} [\text{true } U \text{ protein} > 0]$$

Property 2: “It is certain that protein will not be produced before an RNA molecule is produced”

$$\text{PCTL: } P_{\geq 1} [\neg (\text{protein} > 0) U \text{ RNA} > 0]$$

- $P_{\sim p}[G \psi] \equiv P_{\approx(1-p)}[F \neg\psi]$

where \approx is the logical inverse of \sim which is defined as: $\bar{>} = <$, $\bar{\geq} = \leq$, $\bar{<} = >$, and $\bar{\leq} = \geq$.

Example 5 provides PCTL formulas for verification of the gene expression example (see Section 2.5), with their informal description. In parallel to the previous example (Example 4), for the first property to be true, the model checker will start from one of the initial states, and if there is a path such that the number of protein molecules eventually becomes greater than 0, then the property will hold. The second property indicates that, for the property to hold, in all paths, the number of protein molecules should be zero and remain zero until some RNA molecules are produced.

3.2.4 Continuous Stochastic Logic (CSL)

Continuous Stochastic Logic (CSL) also extends from CTL temporal logic and operates on probabilistic transition systems. CSL is used for specifying probabilistic properties of Continuous Time Markov Chains (CTMCs)—see Definition 11.

For a CTMC $\mathcal{M} = (S, s_i, R, AP, L)$ an infinite path π is a sequence of $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots$ where $R(s_i, s_{i+1}) > 0$ and $t_i \in \mathbb{R}_{>0}$, $\forall i \geq 0$. t_i is the amount of time spent at s_i . If there is an absorbing state s_k such that $k \geq 0$, then the path is finite. π^i means that the starting state of the path is s_i , and the set of the paths of a CTMC model \mathcal{M} starting in state s is denoted by $Path^{\mathcal{M}}(s)$. Additionally, the state occupied at time instant t is denoted as $\pi @ t$. Now we can focus on CSL definition and its semantic. The BNF syntax of CSL is

defined as follows [5, 73]:

$$\begin{aligned}\phi & ::= true \mid a \mid (\phi_1 \wedge \phi_2) \mid \neg\phi \mid P_{\sim p}(\psi) \mid S_{\sim p}(\phi) \\ \psi & ::= X\phi \mid \phi_1 U \phi_2 \mid \phi_1 U^I \phi_2\end{aligned}$$

Similar to PCTL, where a is an atomic proposition, ϕ , ϕ_1 , and ϕ_2 are state formulas, ψ is a path formula, $\sim \in \{<, >, \leq, \geq\}$, $p \in [0, 1]$. I is a non-negative real number, namely $I \in \mathbb{R}_{\geq 0}$ [5, 73]. $\phi_1 U^I \phi_2$ formula asserts that ϕ_1 is true and it will stay true until it reaches a ϕ_2 -true state at some time instant in the interval I . S is steady-state, i.e. long-run, operator and $S_{\sim p}[\phi]$ formula indicates that the steady-state probability of being in a state satisfying ϕ is in the bound $\sim p$. The steady-state is used for querying the behaviour of the system in the long run until the system reaches a balance. For example, when a chemical reaction is in the equilibrium state, which the rate of the forward and reverse reactions are equal so that the concentrations of reactants and products does not change in time, we can use S operator to query the property of the system in this state [73].

Semantics of CSL Let a CTMC model $M = (S, s_i, R, AP, L)$, and $s \in S$, ϕ, ϕ_1, ϕ_2 are CSL state formulas, and ψ be a CSL path formula. The satisfaction relation \models is defined for state formulas by [7, 73]:

- $s \models true$ for all $s \in S$
- $s \models a \iff a \in L(s)$
- $s \models \neg\phi \iff s \not\models \phi$
- $s \models \phi_1 \wedge \phi_2 \iff s \models \phi_1$ and $s \models \phi_2$
- $s \models P_{\sim p}[\psi] \iff Prob^M(s, \psi) \sim p$
- $s \models S_{\sim p}[\phi] \iff \sum_{s' \models \phi} \pi_s^M(s') \sim p$

where:

$$Prob^M(s, \psi) \stackrel{\text{def}}{=} Pr_s\{\pi \in Path^M(s) \mid \pi \models \psi\}$$

and for any path $\pi \in Path^M(s)$

Example 6 *CSL examples for verifying the the gene expression model in Section 2.5.*

Property 1: “The probability that the system eventually produces the protein molecule within first 5 time units is greater than 0”

CSL: $P > 0 [F \leq 5 \text{ protein} > 0]$

Property 2: “It is possible that in long-run the concentration of RNA molecules is less than the concentration of protein molecules.”

CSL: $S > 0 [RNA < \text{protein}]$

- $\pi \models X\phi \iff \pi^1$ is defined and $\pi^1 \models \phi$
- $\pi \models \phi_1 U^I \phi_2 \iff \exists t' \in I. (\pi @ t' \models \phi_2 \wedge \forall t \in [0, t'). (\pi @ t \models \phi_1)$.

Remark 2 *Note that, the logical operator and the temporal operator of CSL can be extended in the same way as we did in PCTL, please see Remark 1.*

Example 6 provides CSL formulas for verification of the gene expression example in Section 2.5, with the informal description of the properties. The formulas are specified based on PRISM language specification [100]. The first property is similar to the first property of Example 5, however, here the number of future states is bounded, therefore, the number of protein should be greater than zero within five steps, on at least one path. Please note that we did not introduce the ‘ F (Eventually)’ operator explicitly, but it can be derived from ‘ U (Until)’ operator, see Remark 1. For the second property to hold, at least for one path, the number of protein molecules should be greater than the number of RNA molecule, and this should be preserved for forever.

We have listed a few property specification languages, but there are several other property languages, such as Probabilistic LTL, PCTL*, and probabilistic LTL with numerical constraints (PLTLc) [7, 40] which are specialised for querying specific type of models. Although the diversity of the dedicated property specification languages is useful for investigating various model features, learning their syntax and understanding the underlying logic is not intuitive. Therefore, especially for non-expert testers such as biologists, it is challenging to know how to formalise the property queries correctly.

Recently, efforts have been made to facilitate property specifications by identifying frequently used property queries, called property patterns or patterns [42, 58]. *Patterns* are commonly represented by natural-like keywords, for example, the reachability of protein molecule property can be expressed as “*eventually* protein > 0”. In Chapter 6 and Chapter 7, we present widely used property patterns and we compare pattern support of different model checking tools.

3.3 Statistical model checking

Statistical model checking (SMC) investigates a subset of the execution paths of the models for model checking. The general idea of SMC is that it compares the number of execution paths on which a property holds with the total number of the execution paths, and tries to statistically infer the satisfaction of the property. Although the verification on the subset of traces is less precise and does not guarantee the correctness of system, the reduced number of paths enables verification of larger models and mitigate the state explosion problem. SMC technique can be seen as a model-independent approach. Because it can be applied to any discrete event system as long as the execution paths of the system can be obtained [107, 120].

A *discrete event system* consists of states, and the system remains in a state for a while before an event triggers a transition from the current state to the same state or to a new state. Therefore, both DTMC and CTMC are discrete event systems. As SMC method is model independent, it is better not to pick any specific model type here, though, DTMC and CTMC definitions can be found in Definition 10 and 11, respectively. Instead, we will introduce the definition of execution paths which can be obtained from any discrete event system [1]. In the following, we will mainly use the definitions given by the two early SMC studies [120] and [107].

Let assume S is the set of the states of a discrete event system \mathcal{M} , where the system occupies some state $s \in S$.

Definition 13 An execution path, π of the a discrete event system, \mathcal{M} , is a sequence

$$\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \dots \quad (3.3)$$

where s_0 is the initial state, and s_i is the system state after i^{th} events, and the time spent in state s_i is t_i . If the k^{th} state is absorbing, then for all $i \geq k$, $s_i = s_k$ and $t_i = \infty$.

For a given discrete event system \mathcal{M} , SMC checks whether a property ψ , e.g. a property specified in a PCTL formula for a DTMC system or a CSL formula for CTMC system, holds on the execution paths of the system. An analysed execution path π provides an observation of a Bernoulli random variable X , which is 1 if the property ψ holds, 0 otherwise [1, 75]. Of course, to make this approach works, the execution paths should be generated in a finite amount time [75].

SMC uses hypothesis testing to determine whether the probability of ψ , i.e. $Pr(\psi)$, is below or above of a threshold θ [1, 75, 107, 118, 120]. Therefore, to determine whether $Pr_{\geq\theta}(\psi)$ holds, we need to check ψ satisfaction on each execution path. Let the ‘null hypothesis’ H_0 , namely what is initially accepted until proved to be otherwise, be $p \geq \theta$, where the probability measure p of ψ is greater than or equal θ , hence $H_0 : p \geq \theta$ [1]. The alternative hypothesis H_1 is the complementary hypothesis of H_0 , which states that p is less than θ , that is $H_1 : p < \theta$. As the problem solution is based on statistical analysis, the result is not guaranteed to be correct; rather we need to know that the statistical test technique has a probability of accepting a wrong hypothesis, however, this is tolerable as long as it is low enough. Now, if we incorrectly reject a true null hypothesis, then we make a *Type I* error which is also called a false positive finding. In other words, we claim H_0 is false and then we accept H_1 , while H_0 is true. If we accept H_0 while it is false and H_1 is true, then we make a *Type II* error, known as the false negative.

The *strength* of the test is determined by the probability of making Type I and Type II errors, which are denoted by α and β , respectively. Ideally, we want to keep them as low as possible. However, this may mean a larger set of samples are required. Also, we should

realise that there might be a trade-off between the error types.

For example, let assume we have a null hypothesis which claims that a cell is healthy, and the alternative hypothesis says the cell is cancer. If SMC erroneously suggests that a cell has cancer, while it is not, this can result in undesirable consequences, such as wrong and expensive treatments. A possible solution to the problem might be being less strict when categorising a cell as cancer if we are not sure, this means we increase the α value. On the other hand, this decision may lead to erroneously categorising some cancer cells as healthy, that β value will increase, which can cause even worse consequences.

The test would have *ideal performance* if the probability of making Type I error is exactly α , and the probability of making Type II error is exactly β . However, getting low probability simultaneously for both error types is very hard, the detail of the problem is explained in [119, 120]. The study [120] proposed relaxing the test by introducing an *indifference region* which boundaries encapsulates θ . In [118] the authors used half-width of the indifference region, denoted as δ , such that $(p - \delta, p + \delta)$ forms the indifference region. Generally, user specifies the δ value. By respecting the indifference region now the acceptance of the null hypothesis $H_0 : p \geq \theta$ changed to $H_0 : p \geq \theta + \delta$, and conversely $H_1 : p < \theta$ is adjusted to $H_1 : p \leq \theta - \delta$ [1, 118]. If p is inside the indifference region a conclusion cannot be driven, namely neither H_0 nor H_1 is true, this generally means that more samples are needed.

In the early work of SMC [120] used the *Wald's Sequential Probability Ratio Test* (SPRT) [115] for accepting hypothesis testing. SPRT does not require a fixed size of the predetermined set of execution paths, instead performs statistical testing on sample collection until it reaches a decision, while also respecting the error probabilities. The following SPRT equation for SMC hypothesis testing is taken from [1].

Let $p_0 = \theta + \delta$, and $p_1 = \theta - \delta$, and the probability of Type I error and Type II errors are α and β , respectively. After gaining m execution paths, i.e. x_1, \dots, x_m that made in the SPRT, let $\sum_{i=1}^m x_i$,

$$f_m = \prod_{i=1}^m \frac{Pr[X_i = x_i | p = p_1]}{Pr[X_i = x_i | p = p_0]} = \frac{P_1^{d_m} (1 - p_1)^{m - d_m}}{P_0^{d_m} (1 - p_0)^{m - d_m}} \quad (3.4)$$

For example (from [1]), when $m = 2$, that is $x_1 = 1$ and $x_2 = 0$; then $Pr[X_1 = x_1 | p = p_1] = p_1$, $Pr[X_2 = x_2 | p = p_1] = 1 - p_1$, and similarly for p_0 , so that $f_m = \frac{p_1(1-p_1)}{p_0(1-p_0)}$. Then, after f_m has been calculated using Equation (3.4), H_0 is accepted if $f_m \leq \beta/(1 - \alpha)$, or H_1 is accepted if $f_m \geq (1 - \beta)/\alpha$, or none of the conditions are satisfied then SPRT moves to next execution path.

There are some other methods used for accepting the hypothesis test, for example [62] used Chernoff-Hoeffding bounds, more recently [67] adopts a Bayesian approach for testing, namely Sequential Hypothesis Testing. We want to refer readers to the following studies for getting broader overview of SMC approaches [1, 75, 118], and Sections 6 and 7 for the details of different SMC tools.

3.4 Model checking tools

Model checking is a powerful method, and there are several tools implemented this approach. In this section, we will summarise some of the popular model checkers which are widely used in academia and industry. We will limit this section by summarising the exact model checkers(which investigates whole state space and provides exact result), especially that are used for deterministic and non-deterministic systems. Detailed information on probabilistic and statistical model checkers, such as PRISM, is provided in Chapter 6 and Chapter 7.

Spin Spin is a widely used open-source model checking tool and is particularly suited for models of multi-threaded software applications and distributed systems, described through interleaved atomic instructions. It features a high-level modelling language to specify systems descriptions, called PROMELA (PROcess MEta LAnguage) [113]. A practical feature of the language is the use of discrete primitive data types and custom data types similar to those in C, allowing fine-grain model details or low-level implementation features to be directly expressed as part of the model [41, 71, 113]. Moreover, Spin provides support for the use of embedded C code, which allows to directly verify the implementation level software specifications. Spin provides complete support for Linear-time Temporal Logic (LTL) and it does not require pre-constructing a global state graph or Kripke structure prior to the search for satisfiability [41, 70, 71, 113].

NuSMV NuSMV is a popular symbolic model checker designed to describe Finite State Machines (FSMs) which range from synchronous to asynchronous, and from the detailed to the abstract systems[79]. NuSMV tool uses NuSMV language which introduces modular hierarchical descriptions and allows the definition of reusable components. NuSMV supports the analysis of specifications expressed in the Computational Tree Logic (CTL) and Linear Temporal Logic (LTL) [9, 79]. It employs symbolic methods, provides a compact representation of the state space by using Binary Decision Diagram (BDD)-based and Boolean Satisfiability Problem (SAT)-based model checking techniques, which increase the efficiency and performance. The tool also has a built-in simulator. It also provides option for generating the execution traces interactively or randomly [9, 79].

nuXmv NUXMV is the successor of NuSMV, and it inherits most of its functionalities. Regarding the input language, NUXMV introduces two new types of variables, namely real and integer. This new feature enables NUXMV to support infinite-state transition systems as well. To analyse infinite-state transition systems NUXMV has employed several new verification algorithms based on Satisfiability Modulo Theory (SMT), more details on the extension of previous algorithms (for finite-state transition systems) and the new

adapted algorithms can be found in NUXMV manual [80]. We also want to note that NUSMV supports both synchronous and asynchronous transition systems, but NUXMV only supports synchronous fair transition systems.

Concerning the usability and functionality, NUXMV includes all the interactive commands provided by NUSMV. Additionally, it supports a set of new commands to enable users to utilise its new features, and the new model checking algorithms for finite-state and infinite-state transition systems [80].

Although the model checking approach relies on mathematically rigorous procedures and formal methods, they may differ in requirements, for example, different model checkers may require different input models and properties expressed for their custom modelling and property specification languages [11]. Therefore, to be able to use different model checkers, scientists need to be familiar with the various modelling and property specification languages. However, describing models in different modelling languages and formulating properties are usually a challenging, cumbersome, error-prone, and time wasting tasks.

3.5 Other analysis methods

3.5.1 Simulation

Simulation is one of the widely-used analysis methods. The simulation of mathematical models involves building a kinetic model of the network and solving the resulting differential equations numerically [106]. Whereas, the simulation of computational models involves obtaining the execution paths of a model. The simulation approach is relatively fast, and it is suitable for analysis of relatively large models. However, typically, it is not possible to exhaustively analyse all computational paths with the simulation. Instead, only a fraction of potential computational paths can be utilised [11]. In non-deterministic and stochastic systems, each state may have more than one successor states, namely different model runs can produce different computational paths. Therefore, some computational paths and model properties may never be revealed, and their conformance to requirements

cannot be determined which makes the simulation approach questionable [11]. In Section 5, we demonstrate the potential of simulation techniques for computational model analysis, by presenting two simulators integrated to the kPWorkbench platform. In addition, numerous simulators have been developed to simulate different type of models, such as, Biocham [23] and Simpathica [3, 32] tools simulate biological systems which are defined as ODEs, and PIPE2 is used for modelling, analysis and simulation of Petri Nets, SPiM [91] is used for simulating stochastic π -calculus models [93], and BioSimWare [20], MeCoSim [90].

3.5.2 Testing

Testing is another widely used method for verifying software systems and computational models. Testing is used for confirming a program fulfils its requirements (does it do what is expected to do?), and also discover program bugs, preferable before its release [112]. While model checking and simulation typically verifies an abstraction of a system, i.e. its model, testing is conducted on the actual software product (of course, the former methods can be applied to actual software programs, and the later can be applied to models, too) [36].

In the testing approach, the key validation technique is executing the system using test data that is produced for certain scenarios (system execution paths) and exploring the corresponding output. However, checking all possible scenarios and potential bugs are generally not possible with testing which leaves doubts if the unexplored paths may contain errors [36]. Therefore, this approach shares same concerns with simulation and statistical model checking as to coverage apply.

Chapter 4

Automating The Model Checking Process

In this chapter, we start with the integrated software platforms which semi-automate the model checking process by employing a high-level language modelling language and demand the users to decide which model checker they prefer to verify their model with. These platforms internally translate the models from the high-level language to the selected model checker specification. Our primary focus will be the NUSMV translator which is an output of this research. In the end, we discuss the shortcomings of such platforms. In the later section, we introduce to the machine learning approach which can be applied for automating the SMC selection that also addresses the drawbacks associated with integrated software suites.

4.1 Integrated software suites

Recently, several state-of-the-art frameworks have been developed to facilitate the modelling and property specification task. They integrate modelling, simulation and model checking on a single platform. Typically, they accommodate more than one simulator and model checker for analysis purpose. The model checkers are not implemented freshly as

part of the tools; instead, they are employed as third-party components. The frameworks receive a computational model and high-level language queries as inputs. After the user selects the desired target model checker for verification, the frameworks translate the provided model and queries to the selected model checker specification and deliver them to the target model checker for verification. [25, 41]. In the following section, we give the details of two frameworks which are developed for modelling, simulation, and analysis of different P system models.

4.1.1 kPWorkbench

kPWorkbench is a software framework that aims to provide a set of tools to facilitate the analysis and formal verification of the kP systems models [41, 72]. The tool translates a kP model specified in kP-Lingua into an internal data structure, where objects represent compartments, multisets of objects, rules, and their connections between compartments (see Figure 4.1). The architecture of the framework allows developers to modify, or extend the internal data structure, and add new model checker translators.

The tool has a custom simulator tool, called kPWorkbench Simulator [8]. It interprets step by step the traces of execution of a kP systems model. The simulator provides a command line user interface displaying the current configuration (the content of each compartment) at each step. The simulations can be output on command prompt or can be redirected to a file [8, 9]. The tool also integrates a third party agent-based modelling framework FLAME [47] used for simulating kP-Lingua specification [8, 9, 37].

In addition, the specifications written in kP-Lingua can be formally verified using Spin model checker. The tool translates the kP system model into Spin model checker specifications. The detailed model translation mechanism is described in [41]. Mainly, due to the space space limitation of Spin, and less support for temporal logics, we have extended the kPWorkbench to accommodate another verification mechanism based on the

Property	Pattern	Language Construct	LTL formula	CTL formula
Next		next p	$X p$	$EX p$
Existence		eventually p	$F p$	$EF p$
Absence		never p	$\neg(F p)$	$\neg(EF p)$
Universality		always p	$G p$	$AG p$
Recurrence		infinitely-often p	$G F p$	$AG EF p$
Steady-State		steady-state p	$F G p$	$AF AG p$
Until		p until q	$p U q$	$A (p U q)$
Response		p followed-by q	$G (p \rightarrow F q)$	$AG (p \rightarrow EF q)$
Precedence		p preceded-by q	$\neg(\neg q U (\neg q \wedge p))$	$\neg(E (\neg q U (\neg q \wedge p)))$

Table 4.1: The LTL and CTL property constructs currently supported by the kP-Queries file (taken from [53]).

NuSMV model checker. Technical details of translation from kP systems into the NuSMV specification are described in the next section.

Recently, the kPWorkbench has been enriched with a new component, called kP-Queries which enable users to query models written in kP-Lingua. The kP-Queries component accommodates a custom editor for specifying properties written in the kP-Queries language. The kP-Queries language is an intuitive and coherent property specification language composed of natural language statements based on predefined property patterns. The kP-Queries component can automatically convert properties specified in the kP-Queries language to CTL and LTL specifications. The kP-Queries component reduces the intricacy of building logical formulas directly in CTL or LTL formalism. The patterns supported by the kP-Queries and corresponding LTL and CTL constructs are listed in Table 4.1. The formal definition and the syntax of the kP-Queries are beyond the scope of this thesis. For more information, please see [53].

4.1.1.1 NuSMV translator

NuSMV translator is developed as a component of kPWorkbench, for translating kP systems model expressed in kP-Lingua into the NuSMV model checker specifications. kP-Workbench automatically verifies the translated NuSMV model by explicitly invoking the

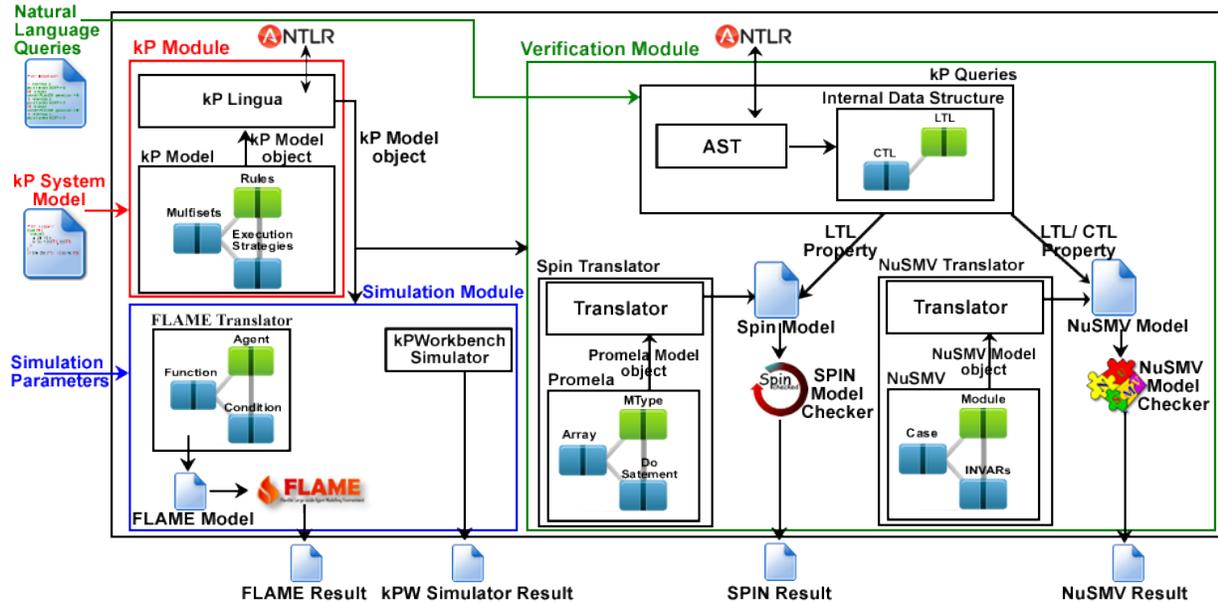


Figure 4.1: kPWorkbench Architecture (taken from [53]).

NuSMV model checker. In this section, we will cover how the models specified in kP-Lingua are translated into the NuSMV specifications.

The translation from a kP system model written in kP-Lingua to a NuSMV model is identified with the following procedure:

Compartment: The kP systems have compartment types and compartments. Compartments are the instances of the compartment types. In NuSMV translation we do not create a NuSMV module for the compartment types, instead, we create a module for each instance of the compartment types, namely for the compartments. The contents of the compartment types, i.e. multisets of objects, rules, execution strategies, are translated as the content of the modules. The initial contents of the kP-Lingua compartments are translated as the parameters of the modules.

Multisets of objects: NuSMV represents models as finite state machines, where the states comprise sets of variables and their associated values (these value sets are required to be bounded), and transitions indicate how inputs cause these values to change. The

multisets of objects are translated as NUSMV variables. Roughly speaking, variables in NUSMV involve in two stages, *declaration* and *assignment*. As its name indicates, the variables are introduced in the declaration stage. NUSMV accepts various variable types, for example, integer, boolean, enumeration, and there are some other types (see [34]). The NUSMV variables should be bounded. That means, they should have a lower and an upper bound, and their bound should be specified during the declaration. Variables are declared after the reserved keyword *VAR*.

Example 7 creates an enumeration, a boolean and an integer variable (it is a snippet from the NUSMV translation of the Example 2, the NUSMV translation is in Appendix A). Some of the NUSMV variables have ‘_’ (underscore) prefix, we used this convention to indicate that the variable is not part of kP-Lingua model, but they are custom variables that we created for achieving kP systems behaviour. The *_status* variable is an enumeration, and it can have one of the three values on the right-hand side. The *_guard1* is a boolean variable, hence, it can have either *true* or *false* values. ‘*a*’ is an integer variable, and it can have integer values from 0 to 5.

Example 7 *NUSMV variable declaration.*

```

VAR
  _status : {_ACTIVE, _willDISSOLVE, _DISSOLVED};
  _guard1 : boolean;
  a : 0 .. 5;

```

NUSMV restricts the upper bound and the lower bound of the integer variables, however, the size of the multisets of objects are not limited in the kP systems. Therefore, the translation has to define an initial bound for each integer variables. Instead of assigning some constant default values to the bounds of the variables, the translator scans the compartments and tries to determine a minimal upper bound value for the variables (indeed, the lower bound is always zero), which is done by following the steps below:

- 1 For each object;
- 2 Set the initial upper bound to 1;

```

3   max1 = the maximum multiplicity of the object from the initial
      multisets of objects;
4   max2 = the maximum multiplicity of the object from the multisets of
      objects of the rewriting rules of the current compartment;
5   max3 = scan the object multiplicity from another compartment:
6       If the current compartment linked to the other compartment, and,
7       if the other compartment has a communication rule and the current
          compartment type is one of the target compartment type of the
          rule, and,
8       if the object exists on the right-hand side of the rule (it is a
          product of the rule),
9       then return the maximum multiplicity of the object from the rule.
10  The new upper bound is the maximum of the max1, max2 and max3.

```

Listing 4.1: Variable upper bound assignment

After calculating the upper bounds of the variables, the NUSMV translator dumps them to an XML file, so that user can change their values.

In the *assignment* stage, the *initial values* of the variables, and the *conditions*, which define the future values of the variables, are specified. The assignment stage starts with *ASSIGN* keyword. The initial values are translated from the multiplicity of the initial multisets of the objects from the kP-Lingua model. If the initial value of a variable is not explicitly set, then a random value within its bound is assigned. The initial value of a variable specified by *init* keyword. For example, *init (_status) := _ACTIVE*, indicates that the *_status* variable starts in the *_ACTIVE* state.

After initialising a variable, the translator assigns a new value based on the satisfaction of some conditions, which are mostly constructed via NUSMV *case statements*. Each *statement* consists of a *condition* and the *value* that it may return on the satisfaction of the condition. Typically, the conditions of the statements are corresponding to the rules of the kP systems. The basic structure of case statement is like the following:

```

case
    condition_1 : value_1;

```

```

    condition_2 : value_2;
        ...
    condition_n : value_n;
esac;

```

The $value_j$ is returned if $j \leq n$, where $n \in \mathbb{N}^+$, such that for all i , $1 \leq i \leq j - 1$, $condition_i$ is false, and $condition_j$ is true. Please note that if there exists a $condition_k$, for $j < k \leq n$ such that $condition_k$ is also true, then the $value_k$ will be ignored, as the case statement always (deterministically) returns only the value of the first ‘true’ condition. Also, at least one condition has to evaluate to true, otherwise it is an error. To prevent this error, the translator adds an extra statement to the end, we call it default statement, which has a constant boolean true condition and, as usual, a value to return. Therefore, all case statements end with (the n^{th} statement) the *default statement*, which syntax is $TRUE : value_n$;

Rules and guards: The rewriting rules are translated as *conditions* of *case statements* inside the modules. If the rewriting rule is a communication rule, then the rule goes into the *main* module of the NUSMV model. The main module is a default NUSMV module where the module instances are initiated and invoked. The membrane dissolution rules are also translated as conditions of the case statements, additionally, a special variable named *_status* is introduced which indicates whether the compartment is active. If the corresponding case statement of a dissolution or a division rule is satisfied (evaluated to true), then the status variable becomes inactive that prevent the module from evolving. The current translator partially supports the membrane division rule, but it does not support the link creation and destruction rules.

The membrane division rule is supported only for the kP system models which should always reach the same final compartment and content set. That is, regardless of the execution order, the systems should always end with the same set of compartments and contents. Even the models which may fulfil this constraint would still require many intermediary variables. However, since the final state of the systems is always going to be the

same, these variables would become extra burdens for the state space. Therefore, computing the final set outside of the model checker state space would reduce the complexity of the problem and it could help us to save some time and memory. We used the C# programming language to externally execute the membrane division rules and obtain the final compartments and contents set of the kP system model. Only the final set is translated to the NuSMV modelling language for further investigation.

Execution Strategies: To implement different execution strategies and to trace their execution order a set of custom variables are introduced. As we explained in Section 2.2, choice, arbitrary, and maximal parallelism strategies execute rules non-deterministically, in NuSMV translation they represented with variables which have names starts with *_cho*, *_arb*, and *_max* prefixes, respectively.

One of the challenges that associated with execution strategies was the non-deterministic selection of the rules. Since NuSMV always selects and returns the first *true* case statement, we had to implement a custom solution to achieve the non-deterministic behaviour. For each rule of a kP system inside an execution strategy, we create a case statement. Therefore, each case statement will have the corresponding case statement of the rule and the default statement. For example, if there are three rules inside a maximal parallelism execution strategy, we need to create one case statement for each rule (three case statements in total), and also each case statement has to have the default statement, too. As each case statement will return a single value, we then combine them in a set by using set *union* operation (specified with *union* keyword), like the following:

```
_max := case
    condition_1 : new_value_1;
    TRUE : new_value_n;
esac union case
    condition_2 : new_value_2;
    TRUE : new_value_n;
esac union case
```

```

    condition_2 : new_value_2;
    TRUE : new_value_n;
esac;

```

By using the set union operation now, we can have more than one *true* statement, and NuSMV selects the set elements non-deterministically [34], and assigns it to *_max* variable. We are close to the solution but not yet, because for each case statement, if the first statement is not *true*, then the default statement will be used, whereas for the other case statements their first statement may evaluate to *true*. This will lead the final set to contain the default statement value as well as the other *true* evaluated case statements' values, and NuSMV will non-deterministically select one of them. This is a problem, because NuSMV may select the value of default statement value, which implicitly means there was no other case statements were *true*, other than the default statement, whereas in fact there were some other conditions were *true*. Therefore, we need to exclude the default statement if there is at least one of the other statements are evaluated to *true*. A natural solution would be using the set *difference* operation to exclude the default statement value when there exists a statement condition other than the default statement evaluates to *true*. Unfortunately, NuSMV set operations do not have the set difference operation. Instead, we used the NuSMV *INVAR* expression, which is used to constrain the set of states (the variable values in our case) based on the evaluation of some conditions. Therefore, we used the *INVAR* expression with case statement like the following:

```

INVAR case
    condition_1 | condition_2 | condition_3 : _max != value_n
    TRUE : _max = value_n;
esac;

```

Please note that '|' is logical *or* operator and, != is *not equal* operator. Therefore, if any of the conditions is *true*, then the result of default statement (*value.n*), cannot be assigned to *_max* value, that means if it is included in the case union operation, now via *INVAR* expression it will be forced to exclude. Therefore, now, NuSMV will non-deterministically

select a value from the set of the *true* conditions such that the value of the default statement is not in the set.

Parallel behaviour from sequential model: The kP system execution strategies may run rules in a maximally parallel way (see Section 2.2). However, the NUSMV models evolve sequentially. To achieve the parallel behaviour, NUSMV translator sequentially executes each compartment rules until all execution strategies are completed. These steps are hidden intermediary steps which are not regarded as the kP systems' steps. Note that, if a compartment completes its intermediary steps earlier, it waits for the other compartments to complete their intermediary steps. Additionally, for each compartment, a copy of its multiset of the objects are created (NUSMV translator adds *_cp* postfix to the name of the copy variables). During the intermediary steps, the rules operate on the original multiset of objects and consume their values, and the copy multisets of objects temporarily accumulate the produced objects. The copy multisets of the objects dumped to the original multisets of objects after all compartments complete the intermediary steps, then the next step is a kP system step. A global boolean variable that belongs to the main module (called *pInS*, in NUSMV translator) is used for indicating the execution is either in intermediary steps, or in kP system steps.

Remark 3 *The network of compartments resembles a graph structure, which could be stored and modified in two-dimensional arrays. However, NUSMV has limited array support. Although it allows defining arrays, it does not allow using a variable index for accessing an array content or assigning a value to it. Additionally, the structure changing rules potentially can be applied infinite times which means we need the dynamic data structure; however, in NUSMV, the array size has to be a constant. The NUSMV array limitations make very difficult to achieve a complete translation from kP systems to NUSMV specifications. Therefore, the link creation and link destruction rules were not possible to be translated, and the division rule partially implemented.*

Remark 4 *We regularly update kPWorkbench website (www.kpworkbench.org) which in-*

cludes the latest version of NUSMV translator accommodated in kPWorkbench. You can also find several cases studies modelled in kP-Lingua and their corresponding NUSMV models.

4.1.2 Infobiotics Workbench

The Infobiotics Workbench (IBW) [66] is another integrated *in silico* platform for modelling, simulation, verification of large-scale systems and synthetic biology models. Additionally, it provides optimization functionality [24]. IBW uses Lattice Population P systems (LPP systems) which allows to model stochastic P systems on 2D geometric lattices [24, 25].

IBW allows simulating models either using deterministic numerical approximation with standard solvers or stochastic simulation. The simulator can execute quite large models which can have thousands of compartments with many reactions and species. Simulation results can be displayed, plotted or exported in text and Excel formats for further analyses [24, 25].

The workbench accommodates two third-party model checkers, PRISM as a numeric model checker and MC2 as a statistical model checker. The tool translates LPP models into selected one of the model checker specification. In addition, similar to kPWorkbench, it accommodates a natural language based specification language component, called Natural Language Query (NLQ). The natural language queries are based on predefined patterns. NLQ internally translates the natural language queries to PRISM and the MC2 property specification language [24, 25]. More information on IBW functionality and architecture is provided in [25].

In this section, we summarised two integrated software suites which are used in the system and synthetic biology domain, but there are also some other web-based or standalone integrated tools, such as Biocham [23], Genetic Network Analyzer (GNA) [57], Taverna

Workbench [14, 101]. Although such platforms significantly decrease the intricacies and complexities of modelling and analysing biological systems, they can only semi-automate the analysing process. Because users still need to know which of the candidate model checker best fits to their needs, and eventually they need to manually select the target model checker, which still requires a significant degree of experience [11]. Therefore, it was highly desirable to have another layer which can automatically match the best model checker to a model and a property query. Machine learning algorithms are suitable candidates (sometimes the only viable choice) for such problems.

4.2 Machine learning

Typically, we implement set of instruction to solve a given task with computers, so-called programming. However, sometimes implementing a program for certain tasks can be hard or even impossible when the problem is complex. For example, maybe we do not have enough expertise in the problem domain, or the solution may require considering too many parameters [109]. Sometimes it is easier to use a statistical approach to gradually improve the performance of such a task with data, that is without explicitly programming an algorithm for the task [102]. This method is called *machine learning*. There are different classes of machine learning algorithms, in this thesis we used supervised machine learning algorithms which works with the data where each sample (example) of the data should have an input and an output pair. In the following, we explain the *supervised machine learning algorithms*, for the other machine learning classes we refer readers to [2, 109].

Supervised machine learning: At a more detailed level, a supervised machine learning algorithm receives a set of samples as input, which is called *training set* and denoted by $X_{train} = (x_i, y_i) \ i \in \mathbb{N}_{\geq 0}$, where (x_i, y_i) is an ordered pair, and x_i is called *input* or *features* and it is represented by a row vector $x_i = [v_1, \dots, v_j]$ where $j \in \mathbb{N}_{\geq 0}$, each element of x , v_j , is a *feature* (i.e. *independent variable*), and y is the *output* variable (i.e. *dependent variable*, or *target variable*), in this thesis we prefer to call it as **target variable**. The size of training

dataset is denoted as $|X_{train}|$. The machine learning algorithms can be grouped further based on the type of the target variable. *Regression algorithms* are used if the target variable, y , is a continuous number, otherwise, if y is a discrete number (or categorical variable), then *classification algorithms* are the appropriate option. The machine learning algorithms start with some initial parameters, and it learns by optimising its parameters based on the samples in the training set [2]. Therefore, we may sometime use the term learner to refer a machine learning algorithm, and use *classifier* if it is trained, i.e. learned. A learned algorithm is basically a function that maps the input to the target variable, therefore, $y_i = f(x_i)$. Since now we have the function f , we can use it for new samples to get their target variables, this process is called prediction.

We may sometimes refer to the fastest SMC prediction study, in Section 7, to explain some machine learning related contexts, hence, please have a quick look before proceeding further. The dataset has 675 biological models ($|X_{total}| = 675$) for prediction of the fastest SMC. Each biological model is a sample, and Figure 4.2 represents one of the samples. The features of the sample contain model properties, such as, number of nodes, vertices, and the target variable is the SMC tool that could verify the corresponding model fastest. Since the target is a categorical variable (its value can be one of five candidate model checkers, namely PRISM, PLASMA-Lab, Ymer, MRMC, and MC2), we can use only the classification algorithms for this problem. Otherwise, if the target was a continuous variable, for example, it could be the execution time of each model checker, then we had to use the regression algorithms.



Figure 4.2: A sample of data for machine learning is an ordered pair of a row vector of features (input) and a target variable (output).

Testing: After training a machine learning algorithm, we will like to know how accurate are its predictions. As the machine learning algorithm is optimised for the training dataset, we cannot use the same dataset for testing, too. Thus, a different dataset should be used for testing, let denote it as X_{test} . The most commonly used measurement for testing the exactitude of a classifier is *accuracy*, which is the ratio of true positive prediction over the total test sample size [89]. We will sometimes use predictive power term to refer the accuracy. Let say we have X_{test} samples for testing, and $n_{test} = |X_{test}|$, the accuracy of a classifier is $\sum_{i=1}^{n_{test}} \frac{I(f(x_i), y_i)}{n_{test}}$, where y_i is the actual target variable and $f(x_i)$ is the predicted target variable, and $I(f(x_i), y_i) = 1$ if $f(x_i) = y_i$, 0 otherwise. That is $(f(x_i), y_i) = 1$ if the i^{th} sample is correctly predicted [89].

Figure 4.3 provides an overview of the stages that the supervised machine learning algorithms involve. The input dataset is a matrix of known features and targets. In the training stage, the supervised machine algorithms learn how to map the features to the targets. After the training, we expect the classifier to identify the target variables the for new, unseen, features.

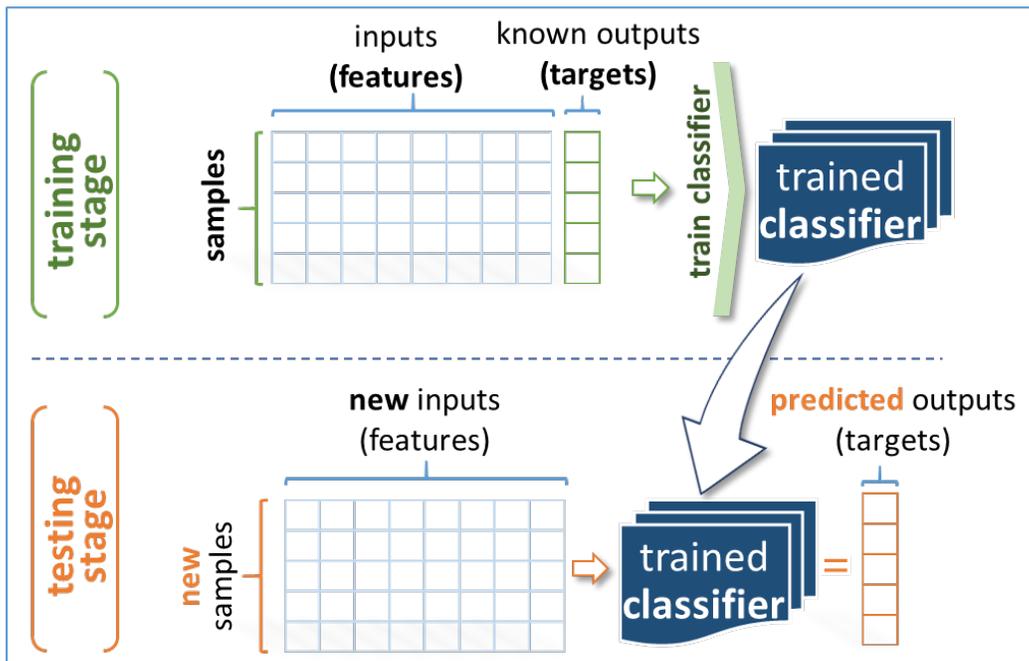


Figure 4.3: An overview of supervised machine learning algorithm.

Cross-validation: Training a machine learning algorithm with more data almost always provides better prediction [39]. Therefore, we would like to use all data for training to improve the accuracy of machine learning algorithms, rather than splitting it into training and test sets. However, as we previously stated, since the algorithm adjusted on the training set we cannot use the same dataset for testing, too, which would probably produce a distorted high accuracy. Fortunately, there is a statistical procedure called cross-validation which enables using the same dataset for training and testing. It divides data into blocks, let's say k blocks, $k - 1$ blocks are used for training and one block is used for testing. In the successive rounds another block is used testing data and the rest of the blocks are used for training. This procedure is repeated until each block used exactly one time for testing, which also means that the process needs to be repeated k times, then the average accuracy of the rounds is regarded as final accuracy. The explained procedure is a special form of cross-validation which is known as k -fold cross-validation. Figure 4.4 demonstrates how k -fold cross validation works. k defines the number of blocks that the dataset is divided. For example, if a dataset segmented into 5 blocks ($k = 5$), it is called as 5-fold cross-validation, or if the number of blocks is 10 ($k = 10$), then it is 10-fold cross-validation, which is the most common form used in data-mining and machine learning studies [97]. We also used the 10-fold cross-validation in Section 7, to assess the accuracy of various machine learning algorithms.

Feature selection “At the end of the day, some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used.” [39]. Identifying important features which affect the target variable at most are crucial for achieving better prediction accuracy. The important features are those not correlated with each other features but well correlated with the target variable. Identification of the important features are easier when the problem domain is well known, and only a small number of parameters are enough to get good accuracy. However, in such cases, we can just program an algorithm for the task, and we will not use the machine learning. However, for more complex cases when many factors determine the target variable, it may

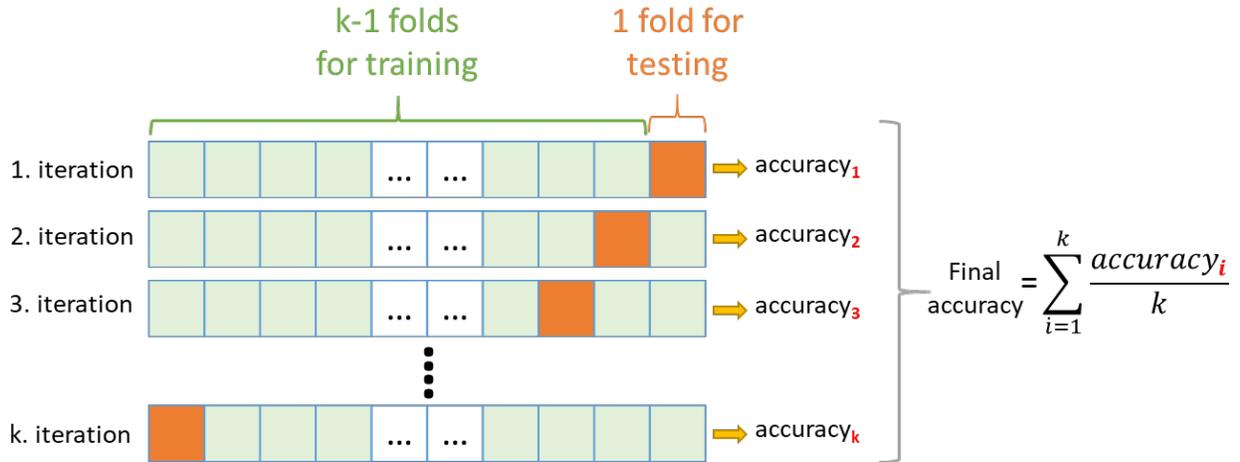


Figure 4.4: k-fold cross-validation.

Key. The green blocks represent training data, and the orange blocks are the testing data. The training and testing data changes after each iteration. After k iteration, all available data is used for both training and testing. For the fastest SMC prediction study, we chose k=10.

not be intuitive identify relevant features. Therefore, to improve the algorithm’s predictive power, one can attempt to provide many features to the algorithm by believing that they can contribute to the predictive power. However, this brings the risk of learning a wrong thing that does not reflect the real relationship between the features and the target. In such a case, the training accuracy can be very high, but the prediction for unseen data can be weak. This problem is called overfitting. Overfitting is a well-known and serious problem in machine learning. One way of reducing the risk of overfitting is keeping the model simple by eliminating the uninformative (less important) features. Feature selection is a broad topic in machine learning, a useful review of various approaches is discussed in [59]. Here we will present a short introduction to the topic and what method we followed for the fastest SMC prediction.

One approach to eliminate less important features or the redundant features could be by brute force, that is trying all subsets of the features and picking the one that gives the highest accuracy. However, it is not hard to see that when the number of the features are high this method is not feasible. Another method is progressively removing a feature at a time and testing whether the prediction accuracy drops, if so, then the feature is important

and it should remain; otherwise, we may consider the feature unimportant or redundant. The drawback of this approach is that sometimes a feature that looks irrelevant in isolation may be relevant in combination [39]. There are also methods that enable feature selection as part of the learner construction process. For example, the depth of tree Decision Trees [27]—predicts a target variable by learning simple rules derived from the data—inherently contains information about the importance of the features, more important features are closer to the root of the tree, and less important are closer to the leaves. Identifying the features regarding the depth of the trees is called *feature importance*. Random forest [26] and Extremely randomized trees (ERT) [50] algorithms internally use the decision trees. Hence, the feature importance algorithm can also be utilised here for feature analysis. In Section 7, we used feature importance algorithm of Extremely randomized trees (ERT) [76] for eliminating less important features. Additionally, we validated the importance of the features by conducting systematic empirical comparisons with and without the features that suggested by ERT feature importance algorithm. We showed that the final feature set we used for the fastest SMC prediction leads very high accuracy. Additionally, they can be used for the fastest stochastic simulation algorithm (SSA) prediction, too (see the Feature Selection part of Section 7).

Machine learning algorithm selection: Usually, different machine learning algorithms can be the best learner in different scenarios. That is, no single machine learning algorithm can be considered as superior to the other learners for all problems. Once the features are identified and the dataset is ready for learning, the systematic empirical comparison of many learners should be conducted. For the fastest SMC prediction, we compared the predictive power of five commonly used machine learning algorithms that are appropriate for classification problems; a support vector classifier (SVM) [33], logistic regression(LR) [121], a nearest neighbour classifier (KNN) [78] and two types of ensemble method, namely, Extremely Randomized Trees (ERT) [50] and Random Forests(RF) [26]. The results showed that ERT and SVM were two most frequently winners. The best classifier for each property pattern type could predict the fastest SMC tool with over 90% accuracy. Therefore,

we employed this two learner in our final product, SMC Predictor (see Section 7, and www.smcpredictor.com).

Part II

The Papers

Chapter 5

Modelling and Analysis

This chapter contains two papers which cover some computational models and their analysis by using simulation and model checking techniques.

First paper: Mehmet E. Bakir, Florentin Ipate, Savas Konur, Laurentiu Mierla, and Ionut Niculescu. Extended simulation and verification platform for kernel P systems. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *Membrane Computing: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers*, pages 158–178. Springer International Publishing, Cham, 2014.

This study provides the definition of membrane computing and its central model, P systems. More specifically it focuses on two subsets of P systems, namely stochastic P systems and kernel P systems. Additionally, it introduces the concepts of stream X-machine and communicating stream X-machine and how they are implemented in the FLAME (Flexible Large-Scale Agent Modelling Environment) simulator. For model analysis, we introduced an initial version of the NuSMV translator tool which can translate kernel P system models to NuSMV model checker specifications. However, the tool was restricted to the non-deterministic choice strategy translation. Later, we improved the NuSMV translator to cover all kernel P system execution strategies. We also introduced the FLAME translator which translates kernel P system models to FLAME specifications. We illustrated how a synthetic biology example can be modelled in kernel P systems, simulated with the FLAME

simulator, and verified with the NuSMV translator.

Second paper: M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High performance simulations of kernel P systems. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICSS)*, pages 409–412, 2014.

The second study highlighted a complementary analysis method to model checking, simulation. Additionally, it examined the performance of a custom *kPWorkbench Simulator* [8] and *FLAME* [37] on a synthetically constructed pulse generator model which is coded as a kernel P system. The result shows that in their default settings a custom kP systems simulator can outperform a general-purpose simulator.

Extended Simulation and Verification Platform for Kernel P Systems

Mehmet E. Bakir¹, Florentin Ipate², Savas Konur¹, Laurentiu Mierla², and Ionut Niculescu³

¹ Department of Computer Science, University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK
{mebakir1,s.konur}@sheffield.ac.uk

² Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro, laurentiu.mierla@gmail.com

³ Department of Computer Science, University of Pitesti
Str. Targul din Vale, nr.1, 110040 Pitesti, Arges, Romania
ionutmihainiculescu@gmail.com

Abstract. *Kernel P systems* integrate in a coherent and elegant manner many of the features of different P system variants, successfully used for modelling various applications. In this paper, we present our initial attempt to extend the software framework developed to support kernel P systems: a formal verification tool based on the NUSMV model checker and a large scale simulation environment based on FLAME. The use of these two tools for modelling and analysis of biological systems is illustrated with a synthetic biology example.

1 Introduction

Membrane computing [16] is a branch of natural computing inspired by the hierarchical structure of living cells. The central model, called *P systems*, consists of a membrane structure, the regions of which contain rewriting rules operating on multisets of objects [16]. P systems *evolve* by repeatedly applying rules, mimicking chemical reactions and transportation across membranes or cellular division or death processes, and halt when no more rules can be applied. The most recent developments in this field are reported in [17].

The origins of P systems make it highly suited as a formalism for representing biological systems, especially (multi-)cellular systems and molecular interactions taking place in different locations of living cells [7]. Different simple molecular interactions or more complex gene expressions, compartment translocation, as well as cell division and death are specified using multiset rewriting or communication rules, and compartment division or dissolution rules. In the case of *stochastic P systems*, constants are associated with rules in order to compute their probabilities and time needed to be applied, respectively, according to the Gillespie algorithm [18]. This approach is based on a Monte Carlo algorithm for the stochastic simulation of molecular interactions taking place inside a single volume or across multiple compartments.

The recently introduced class of *kernel P (kP) systems* [8] integrates in a coherent and elegant manner many of the features of different P system variants, successfully used for modelling various applications. The kP model is supported by a modelling language, called *kP-Lingua*, capable of mapping a kernel P system specification into a machine readable representation. Furthermore, the KPWORKBENCH framework that allows simulation and formal verification of the obtained models using the model checker SPIN was presented in a recent paper [5].

In this paper, we present two new extensions to KPWORKBENCH: a formal verification tool based on the NUSMV model checker [4] and a large scale simulation environment using FLAME (Flexible Large-Scale Agent Modelling Environment) [6], a platform for agent-based modelling on

parallel architectures, successfully used in various applications ranging from biology to macroeconomics. The use of these two tools for modelling and analysis of biological systems is illustrated with a synthetic biology case study, the pulse generator.

The paper is structured as follows. Section 2 defines the formalisms used in the paper, stochastic and kernel P systems as well as stream X-machines and communicating stream X-machine systems, which are the basis of the FLAME platform. Section 3 presents an overview on the kP-lingua language and the simulation and model checking tools. The case study and the corresponding experiments are presented in Section 4 and 5, respectively, while conclusions are drawn in Section 6.

2 Basic definitions

2.1 Stochastic and kernel P systems

Two classes of P systems, used in this paper, will be now introduced. The first model is a **stochastic P system** with its components distributed across a lattice, called *lattice population P systems* [18, 3], which have been applied to some unconventional models e.g. the genetic Boolean gates [13, 12, 19]. For the purpose of this paper we will consider stochastic P systems with only one compartment and the lattice will be regarded as a tissue with some communication rules defined in accordance to its structure.

Definition 1. *A stochastic P system (SP system) with one compartment is a tuple:*

$$SP = (O, M, R) \quad (1)$$

where O is a finite set of objects, called alphabet; M is the finite initial multiset of objects of the compartment, an element of O^* ; R is a set of multiset rewriting rules, of the form $r_k : x \xrightarrow{c_k} y$, where x, y are multisets of objects over O (y might be empty), representing the molecular species consumed (x) and produced (y).

We consider a finite set of labels, L , and a population of SP systems indexed by this family, SP_h , $h \in L$. A lattice, denoted by Lat , is a bi-dimensional finite array of coordinates, (a, b) , with a and b positive integer numbers. Now we can define a lattice population P system, by slightly changing the definition provided in [3].

Definition 2. *A lattice population P system (LPP system) is a tuple*

$$LPP = (Lat, (SP_h)_{h \in L}, Pos, Tr) \quad (2)$$

where Lat, SP_h and L are as above and $Pos : Lat \rightarrow \{SP_h | h \in L\}$ is a function associating to each coordinate of Lat a certain SP system from the given population of SP systems. Tr is a set of translocation rules of the form $r_k : [x]_{h_1} \xrightarrow{c_k} [x]_{h_2}$, where $h_1, h_2 \in L$; this means that the multiset x from the SP system SP_{h_1} , at a certain position in Lat , will move to any of the neighbours (east, west, south, north) in Lat that contains an SP system SP_{h_2} .

The stochastic constant c_k , that appears in both definitions above, is used by Gillespie algorithm [9] to compute the next rule to be applied in the system.

One can see the lattice as a tissue system and the SP systems as nodes of it with some communication rules defined according to the neighbours and also to what they consist of.

Another class of P systems, called **kernel P systems**, has been introduced as a unifying framework allowing to express within the same formalism many classes of P systems [8, 5].

Definition 3. A kernel P system (*kP system*) of degree n is a tuple

$$k\Pi = (O, \mu, C_1, \dots, C_n, i_0) \quad (3)$$

where O is a finite set of objects, called alphabet; μ defines the membrane structure, which is a graph, (V, E) , where V are vertices indicating compartments, and E edges; $C_i = (t_i, w_i)$, $1 \leq i \leq n$, is a compartment of the system consisting of a compartment type from T and an initial multiset, w_i over O ; i_0 is the output compartment where the result is obtained (this will not be used in the paper).

Definition 4. T is a set of compartment types, $T = \{t_1, \dots, t_s\}$, where $t_i = (R_i, \sigma_i)$, $1 \leq i \leq s$, consists of a set of rules, R_i , and an execution strategy, σ_i , defined over $\text{Lab}(R_i)$, the labels of the rules of R_i .

In this paper we will use only one execution strategy, corresponding to the execution of a rule in each compartment, if possible. For this reason the execution strategy will be no longer mentioned in the further definition of the systems. The rules utilised in the paper are defined below.

Definition 5. A rewriting and communication rule, from a set of rules, R_i , $1 \leq i \leq s$, used in a compartment $C_{l_i} = (t_{l_i}, w_{l_i})$, $1 \leq i \leq n$, has the form $x \rightarrow y \{g\}$, where $x \in O^+$ and y has the form $y = (a_1, t_1) \dots (a_h, t_h)$, $h \geq 0$, $a_j \in O$ and t_j indicates a compartment type from T – see Definition 3 – with instance compartments linked to the current compartment, C_{l_i} ; t_j might indicate the type of the current compartment, i.e., t_{l_i} – in this case it is ignored; if a link does not exist (the two compartments are not in E) then the rule is not applied; if a target, t_j , refers to a compartment type that has more than one instance connected to C_{l_i} , then one of them will be non-deterministically chosen.

The definition of a rule from R_i , $1 \leq i \leq s$, is more general than the form provided above, see [8, 5], but in this paper we only use the current form. The guards, denoted by g , are Boolean conditions and their format will be discussed latter on. The guard must be true when a rule is applied.

2.2 X-machines and communicating stream X-machine systems

We now introduce the concepts of stream X-machine and communicating stream X-machine and also discuss how these are implemented in FLAME [6]. The definitions are largely from [11].

A stream X-machine is like a finite automaton in which the transitions are labelled by (partial) functions (called processing functions) instead of mere symbols. The machine has a memory (that represents the domain of the variables of the system to be modelled) and each processing function will read an input symbol, discard it and produce an output symbol while (possibly) changing the value of the memory.

Definition 6. A Stream X-Machine (SXM for short) is a tuple

$Z = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$, where:

- Σ and Γ are finite sets called the input alphabet and output alphabet respectively;
- Q is the finite set of states;
- M is a (possibly) infinite set called memory;
- Φ is the type of Z , a finite set of function symbols. A basic processing function $\phi : M \times \Sigma \rightarrow \Gamma \times M$ is associated with each function symbol ϕ .
- F is the (partial) next state function, $F : Q \times \Phi \rightarrow 2^Q$. As for finite automata, F is usually described by a state-transition diagram.
- I and T are the sets of initial and terminal states respectively, $I \subseteq Q, T \subseteq Q$;

- m_0 is the initial memory value, where $m_0 \in M$;
- all the above sets, i.e., $\Sigma, \Gamma, Q, M, \Phi, F, I, T$, are non-empty.

A configuration of a SXM is a tuple (m, q, s, g) , where $m \in M, q \in Q, s \in \Sigma^*, g \in \Gamma^*$. An initial configuration will have the form (m_0, q_0, s, ϵ) , where m_0 is as in Definition 6, $q_0 \in I$ is an initial state, and ϵ is the empty word. A final configuration will have the form (m, q_f, ϵ, g) , where $q_f \in T$ is a terminal state. A change of configuration, denoted by $\vdash, (m, q, s, g) \vdash (m', q', s', g')$, is possible if $s = \sigma s'$ with $\sigma \in \Sigma, g' = g\gamma$ with $\gamma \in \Gamma$ and there exists $\phi \in \Phi$ such that $q' \in F(q, \phi)$ and $\phi(m, \sigma) = (\gamma, m')$. A change of configuration is called a *transition* of a SXM. We denote by \vdash^* the reflexive and transitive closure of \vdash .

A number of communicating SXMs variants have been defined in the literature. In what follows we will be presenting the communicating SXM model as defined in [11] since this is the closest to the model used in the implementation of FLAME [6] (there are however, a few differences that will be discussed later). The model defined in [11] appears to be also the most natural of the existing models of communicating SXMs since each communicating SXM is a standard SXM as defined by Definition 6. In this model, each communicating SXM has only one (global) input stream of inputs and one (global) stream of outputs. Depending on the value of the output produced by a communicating SXM, this is placed in the global output stream or is processed by a SXM component. For a more detailed discussion about the differences between various models of communicating SXMs see [15].

The following definitions are largely from [11].

Definition 7. A Communicating Stream X-Machine System (CSXMS for short) with n components is a tuple $S_n = ((Z_i)_{1 \leq i \leq n}, E)$, where:

- $Z_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i,0})$ is the SXM with number $i, 1 \leq i \leq n$.
- $E = (e_{ij})_{1 \leq i, j \leq n}$ is a matrix of order $n \times n$ with $e_{ij} \in \{0, 1\}$ for $1 \leq i, j \leq n, i \neq j$ and $e_{ii} = 0$ for $1 \leq i \leq n$.

A CSXMS works as follows:

- Each individual *Communicating SXM* (CSXM for short) is a SXM plus an implicit input queue (i.e., of FIFO (first-in and first-out) structure) of infinite length; the CSXM only consumes the inputs from the queue.
- An input symbol σ received from the external environment (of FIFO structure) will go to the input queue of a CSXM, say Z_j , provided that it is contained in the input alphabet of Z_j . If more than one such Z_j exist, then σ will enter the input queue of one of these in a non-deterministic fashion.
- Each pair of CSXMs, say Z_i and Z_j , have two FIFO channels for communication; each channel is designed for one direction of communication. The communication channel from Z_i to Z_j is enabled if $e_{ij} = 1$ and disabled otherwise.
- An output symbol γ produced by a CSXM, say Z_i , will pass to the input queue of another CSXM, say Z_j , providing that the communication channel from Z_i to Z_j is enabled, i.e. $e_{ij} = 1$, and it is included in the input alphabet of Z_j , i.e. $\gamma \in \Sigma_j$. If these conditions are met by more than one such Z_j , then γ will enter the input queue of one of these in a non-deterministic fashion. If no such Z_j exists, then γ will go to the output environment (of FIFO structure).
- A CSXMS will receive from the external environment a sequence of inputs $s \in \Sigma^*$ and will send to the output environment a sequence of outputs $g \in \Gamma^*$, where $\Sigma = \Sigma_1 \cup \dots \cup \Sigma_n$, $\Gamma = (\Gamma_1 \setminus In_1) \cup \dots \cup (\Gamma_n \setminus In_n)$, with $In_i = \cup_{k \in K_i} \Sigma_k$, and $K_i = \{k \mid 1 \leq k \leq n, e_{ik} = 1\}$, for $1 \leq i \leq n$.

A *configuration* of a CSXMS S_n has the form $z = (z_1, \dots, z_n, s, g)$, where:

- $z_i = (m_i, q_i, \alpha_i, \gamma_i)$, $1 \leq i \leq n$, where $m_i \in M_i$ is the current value of the memory of Z_i , $q_i \in Q_i$ is the current state of Z_i , $\alpha_i \in \Sigma_i^*$ is the current contents of the input queue and $\gamma_i \in \Gamma_i^*$ is the current contents of the output of Z_i ;
- s is the current value of the input sequence;
- g is the current value of the output sequence.

An *initial configuration* has the form $z_0 = (z_{1,0}, \dots, z_{n,0}, s, \epsilon)$, where $z_{i,0} = (m_{i,0}, q_{i,0}, \epsilon, \epsilon)$, with $q_{i,0} \in I_i$. A *final configuration* has the form $z_f = (z_{1,f}, \dots, z_{n,f}, \epsilon, g)$, where $z_{i,f} = (m_i, q_{i,f}, \alpha_i, \gamma_i)$, with $q_{i,f} \in T_i$.

A change of configuration happens when at least one of the X-machines changes its configuration, i.e., a processing function is applied. More formally, a change of configuration of a CSXMS S_n , denoted by \models ,

$$z = (z_1, \dots, z_n, s, g) \models z' = (z'_1, \dots, z'_n, s', g'),$$

with $z_i = (m_i, q_i, \alpha_i, \gamma_i)$ and $z'_i = (m'_i, q'_i, \alpha'_i, \gamma'_i)$, is possible if one of the following is true for some i , $1 \leq i \leq n$:

1. $(m'_i, q'_i, \alpha'_i, \gamma'_i) = (m_i, q_i, \alpha_i \sigma, \epsilon)$, with $\sigma \in \Sigma_i$; $z'_k = z_k$ for $k \neq i$; $s = \sigma s'$, $g' = g$;
2. $(m_i, q_i, \sigma \alpha_i, \gamma_i) \vdash (m'_i, q'_i, \alpha'_i, \gamma)$ with $\sigma \in \Sigma_i$, $\gamma \in (\Gamma_i \setminus In_i)$; $z'_k = z_k$ for $k \neq i$; $s' = s$, $g' = g\gamma$;
3. $(m_i, q_i, \sigma \alpha_i, \gamma_i) \vdash (m'_i, q'_i, \alpha'_i, \gamma)$ with $\sigma \in \Sigma_i \cup \{\epsilon\}$, $\gamma \in (\Gamma_i \cap \Sigma_j) \cup \{\epsilon\}$ for some $j \neq i$ such that $e_{ij} = 1$; $(m'_j, q'_j, \alpha'_j, \gamma'_j) = (m_j, q_j, \alpha_j \gamma, \epsilon)$; $z'_k = z_k$ for $k \neq i$ and $k \neq j$; $s' = s$, $g' = g$;

A change of configuration is called a *transition* of a CSXMS. We denote by \models^* the reflexive and transitive closure of \models .

The correspondence between the input sequence applied to the system and the output sequence produced gives rise to the *relation computed by the system*, f_{S_n} . More formally, $f_{S_n} : \Sigma \longleftrightarrow \Gamma$ is defined by: $s f_{S_n} g$ if there exists $z_0 = (z_{1,0}, \dots, z_{n,0}, s, \epsilon)$ and $z_f = (z_{1,f}, \dots, z_{n,f}, \epsilon, g)$ an initial and final configuration, respectively, such that $z_0 \models^* z_f$ and there is no other configuration z such that $z_f \models z$.

In [15] it is shown that for any kP system, $k\Pi$, of degree n , $k\Pi = (O, \mu, C_1, \dots, C_n, i_0)$, using only rewriting and communication rules, there is a communicating stream X-machine system, $S_{n+1} = ((Z_{i,t_i})_{1 \leq i \leq n}, Z_{n+1}, E')$ with $n+1$ components such that, for any multiset w computed by $k\Pi$, there is a complete sequence of transitions in S_{n+1} leading to $s(w)$, the sequence corresponding to w . The first n CSXM components simulate the behaviour of the compartment C_i and the $(n+1)$ th component Z_{n+1} helps synchronising the other n CSXMs. The matrix $E' = (e'_{i,j})_{1 \leq i,j \leq n+1}$ is defined by: $e'_{i,j} = 1$, $1 \leq i, j \leq n$, iff there is an edge between i and j in the membrane structure of $k\Pi$ and $e'_{i,n+1} = e'_{n+1,i} = 1$, $1 \leq i \leq n$ (i.e., there are connections between any of the first n CSXMs and Z_{n+1} , and vice-versa). Only one input symbol σ_0 is used; this goes into the input queue of Z_{n+1} , which, in turn, sends $[\sigma_0, i]$ to each CSXM Z_i and so initializes their computation, by processing the strings corresponding to their initial multisets. Each computation step in $k\Pi$ is reproduced by a number of transitions in S_{n+1} . Finally, when the kP system stops the computation, and the multiset w is obtained in C_{i_0} , then S_{n+1} moves to a final state and the result is sent out as an output sequence, $s(w)$.

We now briefly discuss the implementation of CSXMSs in FLAME. Basically, there are two restrictions that the FLAME implementation places on CSXMSs: (i) the associated FA of each CSXM has no loops; and (ii) the CSXMSs receive no inputs from the environment, i.e., the inputs received are either empty inputs or outputs produced (in the previous computation step) by CSXM components of the system. As explained above, a kP system is transformed into a communicating X-machine system by constructing, for each membrane, a communicating X-machine that simulates its behaviour; an additional X-machine, used for the synchronization of the others, is also used. In FLAME, however, the additional X-machine is no longer needed since the synchronization is achieved through message passing - for more details see Section 3.1 and Appendix.

3 Tools used for kP system models

The kP system models are specified using a machine readable representation, called *kP-Lingua* [5]. A slightly modified version of an example from [5] is presented below, showing how various kP systems concepts are represented in kP-Lingua.

Example 1. A type definition in kP-Lingua.

```

type C1 {
  choice {
    > 2b : 2b -> b, a(C2) .
    b -> 2b .
  }
}
type C2 {
  choice {
    a -> a, {b, 2c}(C1) .
  }
}
m1 {2x, b} (C1) - m2 {x} (C2) .

```

Example 1 shows two compartment types, *C1*, *C2*, with corresponding instances *m1*, *m2*, respectively. The instance *m1* starts with the initial multiset $2x, b$ and *m2* with an x . The rules of *C1* are selected non-deterministically, only one at a time. The first rule is executed only when its guard is true, i.e., only when the current multiset has at least three b 's. This rule also sends an a to the instance of the type *C2* linked to it. In *C2*, there is only a rule which is executed only when there is an a in the compartment.

The specifications written in kP-Lingua can be simulated and formally verified using a model checker called SPIN. In this paper, we show two further extensions, another verification mechanism based on the NUSMV model checker [4] and a large scale simulation environment using FLAME [6]. These two tools are integrated into KPWORKBENCH, which can be downloaded from the KPWORKBENCH web page [14].

3.1 Simulation

The ability of simulating kernel P systems is one important aspect provided by a set of tools supporting this formalism. Currently, there are two different simulation approaches (a performance comparison can be found in [1]). Both receive as input a kP-Lingua model and outputs a trace of the execution, which is mainly used for checking the evolution of a system and for extracting various results out of the simulation.

KPWorkbench Simulator. KPWORKBENCH contains a simulator for kP system models and is written in the C# language. The simulator is a command line tool, providing a means for configuring the traces of execution for the given model, allowing the user to explicitly define the granularity of the output information by setting the values for a concrete set of parameters:

- *Steps* - a positive integer value for specifying the maximum number of steps the simulation will run for. If omitted, it defaults to 10.
- *SkipSteps* - a positive integer value representing the number of steps to skip the output generation. By using this parameter, the simulation trace will be generated from the step next to the currently specified one, onward. If not set, the default value is 0.

- *RecordRuleSelection* - defaulting to *true*, takes a boolean value on which to decide if the rule selection mechanism defined by the execution strategy will be generated into the output trace.
- *RecordTargetSelection* - if *true* (which is also the default value), traces the resolution of the communicating rules, outputting the non-deterministically selected membrane of a specified type to send the objects to.
- *RecordInstanceCreation* - defaulting to *true*, specifies if the membrane creation processes should be recorded into the output simulation trace.
- *RecordConfigurations* - if *true* (being also the default setting), generates as output, at the end of a step, the multiset of objects corresponding to each existing membrane.
- *ConfigurationsOnly* - having precedence over the other boolean parameters, sets the value of the above flags to *false*, except the one of the *RecordConfigurations*, causing the multiset configuration for each of the existing membranes to be the only output into the simulation trace. The default value is *false*.

FLAME-based Simulator. The agent-based modeling framework FLAME can be used to simulate kP-Lingua specifications. One of the main advantages of this approach is the high scalability degree and efficiency for simulating large scale models.

A general FLAME simulation requires the provision of a model for specifying the agents representing the definitions of communicating X-machines, whose behaviour is to be simulated, together with the input data representing the initial values of the memory for the generated X-machines. The model specification is composed of an xml file defining the structure of the agents, while their behaviour is provided as a set of functions in the C programming language.

In order to be able to simulate kernel P system models using the FLAME framework, an automated model translation has been implemented for converting the kP-Lingua specification into the above mentioned formats. Thus, the various compartments defined into the kP-Lingua model are translated into agent definitions, while the rule execution strategies corresponds to the transitions describing the behaviour of the agents. More specifically, each membrane of the kP system is represented by an agent. The rules are stored together with the membrane multiset as agent data. For each type of membrane from the kP system, a type of agent is defined, and for each execution strategy of the membrane, states are created in the X-machine. Transitions between the two states are represented by C functions that are executed in FLAME when passing from one state to another. Each type of strategy defines a specific function that applies the rules according to the execution strategy. A detailed description of the algorithm for translating a kP system into FLAME is given in the Appendix.

Each step of the simulation process modifies the memory of the agents, generating at the same time output xml files representing the configuration of the corresponding membranes at the end of the steps. The granularity level of the information defining the simulation traces is adjustable by providing a set of concrete parameters for the input data set.

3.2 Model Checking

KPWORKBENCH already integrates the SPIN model checker [10]. A more detailed account can be found in [5]. In this paper, we also integrate the NUSMV model checker [4] to the KPWORKBENCH platform to be able to verify branching-time semantics. NUSMV is designed to verify synchronous and asynchronous systems. Its high-level modelling language is based on *Finite State Machines* (FSM) and allows the description of systems in a modular and hierarchical manner. NUSMV supports the analysis of specification expressed in *Linear-time Temporal Logic* (LTL) and *Computation Tree Logic* (CTL). NUSMV employs *symbolic* methods, allowing a compact representation of the state space to increase the efficiency and performance. The tool also permits conducting

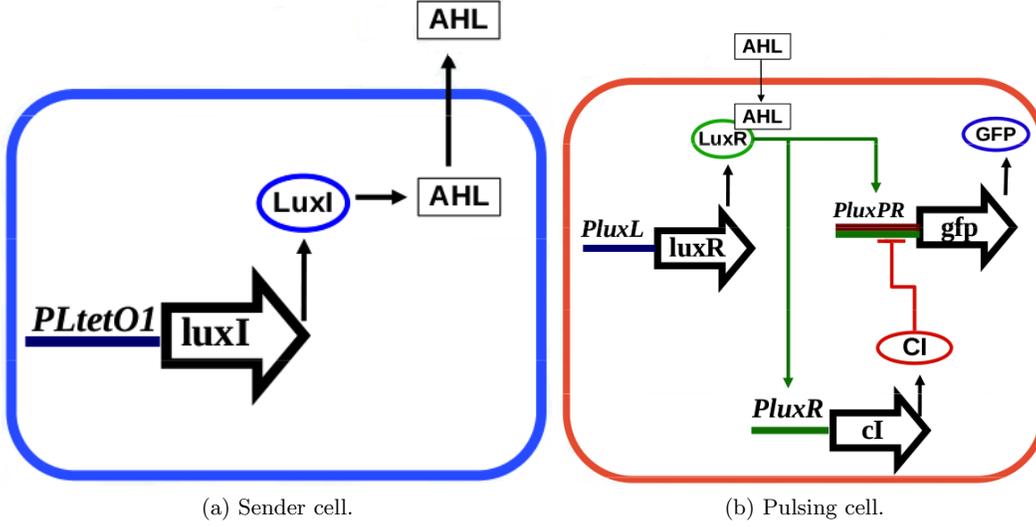


Fig. 1: Two cell types of the pulse generator system (taken from [3]).

simulation experiments over the provided FSM model by generating traces either interactively or randomly.

We note that the NuSMV tool is currently considered for a restricted subset of the kP-Lingua language, and we only consider one execution strategy, *nondeterministic choice*.

4 Case study: Pulse Generator

The *pulse generator* [2] is a synthetic biology system, which was analysed stochastically in [3, 13]. It is composed of two types of bacterial strains: *sender* and *pulsing* cells (see Figure 1). The sender cells produce a signal (30C6-HSL) and propagates it through the pulsing cells, which express the green fluorescent protein (GFP) upon sensing the signal. The excess of the signalling molecules are propagated to the neighbouring cells.

Sender cells synthesize the signalling molecule 30C6-HSL (AHL) through the enzyme LuxI, expressed under the constitutive expression of the promoter PLtet01. Pulsing cells express GFP under the regulation of the PluxPR promoter, activated by the LuxR_30C6_2 complex. The LuxR protein is expressed under the control of the PluxL promoter. The GFP production is repressed by the transcription factor CI, codified under the regulation of the promoter PluxR that is activated upon binding of the transcription factor LuxR_30C6_2.

4.1 Stochastic model

The formal model consists of two bacterial strains, each one is represented by an SP system model. So, $L = \{sender, pulsing\}$, describes these two labels and accordingly:

$$SP_h = (O_h, M_h, R_h), h \in L \quad (4)$$

where

$$O_{sender} = \{PLtet01_geneLuxI, proteinLuxI, proteinLuxI_Rib, rnaLuxI, rnaLuxI_RNAP, signal30C6\}$$

$$M_{sender} = PLtet01_geneLuxI$$

$$\begin{aligned}
 R_{sender} = \{ & r_1 : \text{PLtet01.geneLuxI} \xrightarrow{k_1} \text{PLtet01.geneLuxI} + \text{rnaLuxI.RNAP} \quad k_1 = 0.1, \\
 & r_2 : \text{rnaLuxI.RNAP} \xrightarrow{k_2} \text{rnaLuxI} \quad k_2 = 3.36, \\
 & r_3 : \text{rnaLuxI} \xrightarrow{k_3} \text{rnaLuxI} + \text{proteinLuxI.Rib} \quad k_3 = 0.0667, \\
 & r_4 : \text{rnaLuxI} \xrightarrow{k_4} \quad k_4 = 0.004, \\
 & r_5 : \text{proteinLuxI.Rib} \xrightarrow{k_5} \text{proteinLuxI} \quad k_5 = 3.78, \\
 & r_6 : \text{proteinLuxI} \xrightarrow{k_6} \quad k_6 = 0.067, \\
 & r_7 : \text{proteinLuxI} \xrightarrow{k_7} \text{proteinLuxI} + \text{signal30C6} \quad k_7 = 5 \}
 \end{aligned}$$

and

$$\begin{aligned}
 O_{pulsing} = \{ & \text{CI2, LuxR2, PluxL.geneLuxR, PluxPR.CI2.geneGFP,} \\
 & \text{PluxPR.LuxR2.CI2.geneGFP, PluxPR.LuxR2.geneGFP, PluxPR.geneGFP,} \\
 & \text{PluxR.LuxR2.geneCI, PluxR.geneCI, proteinCI, proteinCI.Rib, proteinGFP,} \\
 & \text{proteinGFP.Rib, proteinLuxR, proteinLuxR.30C6, proteinLuxR.Rib, rnaCI,} \\
 & \text{rnaCI.RNAP, rnaGFP, rnaGFP.RNAP, rnaLuxR, rnaLuxR.RNAP, signal30C6} \}
 \end{aligned}$$

$$M_{pulsing} = \text{PluxL.geneLuxR, PluxR.geneCI, PluxPR.geneGFP.}$$

The set of rules ($R_{pulsing}$) is presented in Table 6 (Appendix).
The translocation rules are:

$$\begin{aligned}
 Tr = \{ & r_1 : [\text{signal30C6}]_{sender} \xrightarrow{k_1} [\text{signal30C6}]_{pulsing} \quad k_1 = 1.0, \\
 & r_2 : [\text{signal30C6}]_{sender} \xrightarrow{k_2} [\text{signal30C6}]_{sender} \quad k_2 = 1.0, \\
 & r_3 : [\text{signal30C6}]_{pulsing} \xrightarrow{k_3} [\text{signal30C6}]_{pulsing} \quad k_3 = 1.0 \}.
 \end{aligned}$$

The lattice, given by Lat , is an array with n rows and m columns of coordinates (a, b) , where $0 \leq a \leq n - 1$ and $0 \leq b \leq m - 1$. The values n and m will be specified for various experiments conducted in this paper. If we assume that the first column is associated with *sender* SP systems and the rest with *pulsing* systems, we formally express this as follows: $Pos(a, 0) = SP_{sender}$, $0 \leq a \leq n - 1$, and $Pos(a, b) = SP_{pulsing}$, $0 \leq a \leq n - 1$ and $1 \leq b \leq m - 1$.

4.2 Nondeterministic model

Non-deterministic models are used for qualitative analysis. They are useful for detecting the existence of molecular species rather than for measuring their concentration. A typical non-deterministic model can be obtained from a stochastic model by removing the kinetics constants.

More precisely, one can define two types corresponding to the two bacterial strains in accordance with Definition 4, namely $T = \{sender, pulsing\}$, and the corresponding rule sets, R'_{sender} and $R'_{pulsing}$. The rules from R'_{sender} are obtained from R_{sender} and $r_1, r_2 \in Tr$, and those from $R'_{pulsing}$ are obtained from $R_{pulsing}$ and $r_3 \in Tr$, by removing the kinetic rates. For each rule from the set Tr , namely $r_k : [x]_{h_1} \xrightarrow{c_k} [x]_{h_2}$, the corresponding rule of the kP system will be $r_k : x \rightarrow x(t)$, where $t \in T$. The execution strategies are those described in the associated definitions of the kP systems.

The kP system with $n \times m$ components is given, in accordance with Definition 3, by the graph with vertices $C_{a,b} = (t_{a,b}, w_{a,b})$, where $t_{a,b} \in T$ and $w_{a,b}$ is the initial multiset, $0 \leq a \leq n - 1$, $0 \leq b \leq m - 1$; and edges where each component $C_{a,b}$, with $0 \leq a \leq n - 1$, $0 \leq b \leq m - 2$, is connected to its east neighbor, $C_{a,b+1}$, and each component $C_{a,b}$, with $0 \leq a \leq n - 2$, $0 \leq b \leq m - 1$ is connected to the south neighbor, $C_{a+1,b}$. The types of these components are $t_{a,0} = sender$, $0 \leq a \leq n - 1$, and $t_{a,b} = pulsing$, $0 \leq a \leq n - 1$ and $1 \leq b \leq m - 1$. The initial multisets are $w_{a,0} = M_{sender}$, $0 \leq a \leq n - 1$, and $w_{a,b} = M_{pulsing}$, $0 \leq a \leq n - 1$ and $1 \leq b \leq m - 1$.

So, one can observe the similitude between the lattice and function Pos underlying the definition of the LPP system and the graph and the types associated with the kP system.

4.3 Nondeterministic simplified model

In order to relieve the state explosion problem, models can also be simplified by replacing a long chain of reactions by a simpler rule set which will capture the starting and ending parts of this chain, and hence eliminating species that do not appear in the new rule set. With this transformation we achieve a simplification of the state space, but also of the number of transitions associated with the model.

The non-deterministic system with a set of compacted rules for the sender cell is obtained from the kP system introduced above and consists of the same graph with the same types, T , and initial multisets, $w_{a,b}$, $0 \leq a \leq n - 1$, $0 \leq b \leq m - 1$, but with simplified rule sets obtained from R'_{sender} and $R'_{pulsing}$, denoted R''_{sender} and $R''_{pulsing}$, respectively, where R''_{sender} is defined as follows:

$$R''_{sender} = \{r_1 : \text{PLtet01_geneLuxI} \rightarrow \text{PLtet01_geneLuxI} + \text{rnaLuxI_RNAP}, \\ r_2 : \text{proteinLuxI} \rightarrow, \\ r_3 : \text{proteinLuxI} \rightarrow \text{proteinLuxI} + \text{signal30C6}, \\ r_4 : \text{signal30C6} \rightarrow \text{signal30C6} \text{ (pulsing)}, \\ r_5 : \text{signal30C6} \rightarrow \text{signal30C6} \text{ (sender)}\}$$

and $R''_{pulsing}$ is defined in Table 7 (Appendix).

5 Experiments

5.1 Simulation

The simulation tools have been used to check the temporal evolution of the system and to infer various information from the simulation results. For a kP system of 5×10 components, which comprises 25 sender cells and 25 pulsing cells, we have observed the production and transmission of the signalling molecules from the sender cells to the furthest pulsing cell and the production of the green florescent protein.

FLAME Results. As explained before, in FLAME each agent is represented by an X-machine. When an X-machine reaches its final state, the data is written to the hard disk and then used as input for the next iteration. Since the volume of data increases with the number of membranes, the more membranes we have, the more time for reading and writing the data (from or to the hard disk) is required. Consequently, when the number of membranes is large, the time required by the read and write operations increases substantially, so the simulation may become infeasible⁴. For example, for the pulsing generator system it was difficult to obtain simulation results after 100,000 steps; the execution time for 100,000 steps was approximately one hour.

The signalling molecule `signal30C6` appeared for the first time in the sending cell `sender1,1` at the 27th step; after that, it appeared and disappeared many times. In the pulsing cell `pulse5,9`, the signalling molecule appeared for the first time at 4963 steps, while `proteinGFP` was produced for the first time after 99,667 steps.

The results of the FLAME simulation show that the signaling molecules produced in the sending cells are propagated to the pulsating cells which, in turn, produce `proteinGFP`. The results of the simulation are given in Table 1. In 100,000 steps (the maximum number of steps considered for the FLAME simulation), the farthest cell in which `proteinGFP` was produced was `pulse5,9` - this was produced after 99,667 steps.

⁴ On the other hand, the distributed architecture of FLAME allows the simulation to be run on parallel supercomputers with great performance improvements, but this is beyond the scope of this paper.

Table 1: FLAME results.

Step Interval	sender _{1,1}		pulse _{5,9}
	signal30C6	signal30C6	proteinGFP
0 – 10,000	Exist	Exist	None
10,001 – 20,000	Exist	Exist	None
20,001 – 30,000	Exist	Exist	None
30,001 – 40,000	Exist	Exist	None
40,001 – 50,000	Exist	Exist	None
50,001 – 60,000	Exist	Exist	None
60,001 – 70,000	Exist	Exist	None
70,001 – 80,000	Exist	Exist	None
80,001 – 90,000	Exist	Exist	None
90,001 – 99,666	Exist	Exist	None
99,667 – 100,000	Exist	Exist	Exist

Table 2: KPWORKBENCH SIMULATOR results.

Step Interval	sender _{1,1}		pulse _{5,10}	
	signal30C6	signal30C6	proteinGFP	proteinGFP
0 – 300,000	Exist	Exist	None	None
300,001 – 600,000	Exist	Exist	None	None
600,001 – 900,000	Exist	Exist	None	None
900,001 – 1,200,000	Exist	Exist	None	None
1,200,001 – 1,500,000	Exist	Exist	None	None
1,500,001 – 1,800,000	Exist	Exist	None	None
1,800,001 – 2,100,000	Exist	Exist	None	None
2,100,001 – 2,400,000	Exist	Exist	None	None
2,400,001 – 2,700,000	Exist	Exist	Exist	Exist
2,700,001 – 3,000,000	Exist	Exist	None	None

KPWorkbench Simulator Results. KPWORKBENCH SIMULATOR is a specialised simulation tool and provides better results, in terms of execution time, than a general purpose simulation environment like FLAME. This is mainly due to the fact that this approach makes the simulation to be performed in a single memory space, that scales according to the number of membranes used in the model and the number of objects resulting from applying the rules in each simulation step.

Table 2 presents the production and availability of the signaling molecule at the first sender cell (i.e. **sender_{1,1}**) and the transmission of the signaling molecule and the production of the green fluorescent protein at the furthest pulsing cell (i.e. **pulse_{5,10}**).

We have run the simulator for 3,000,000 time steps. The **sender_{1,1}** cell was able to produce the signaling molecule at 22 steps, and later produced more signaling molecules. The **pulse_{5,10}** cell, as the furthest pulse generator cell, was able to receive the signaling molecule at the 5474 step. But, the production of **proteinGFP** was possible at the 2,476,813 step, and it remained inside the cell until the 2,476,951 step, then it was consumed.

The simulation results show that the signaling molecule can be produced and transmitted by a sender cell. In addition, a pulse generator cell can have a signaling molecule only after a sender cell sends it, and can use the signal for the production of **proteinGFP** in later steps.

Table 3: Property patterns used in the verification experiments.

Prop.	Informal specification and the corresponding CTL translations
1	<i>X will eventually be produced.</i> $EF X>0$
2	<i>The availability/production of X will eventually lead to the production of Y.</i> $AG (X \Rightarrow EF Y)$
3	<i>Y cannot be produced before X is produced.</i> $\neg E (X=0 U Y>0)$

5.2 Verification

The properties of the system are verified using the NUSMV model checker, fully integrated into the KPWORKBENCH platform. In this section, we first verify individual cell properties, and then verify the properties of the whole kP system, involving multiple cells that are distributed within the lattice and interact with each other via the translocation rules.

Our verification results show that when the cell population is small, the properties can be verified using reasonable computational resources. However, given that the complete rule set is used, when the number of cell increases, verification becomes no longer feasible due to the state explosion problem. To mitigate this problem, we have used a simplified rule set to verify the cell interaction properties when the cell population is large.

Complete Rule Sets. Experiments under this section are conducted on a small population of multi-cellular systems including the complete set of rules. We have analysed two pulse-generator systems that differ only in the number of pulse generator cells. The first group consists of one sender cell and one pulse generator cell, i.e. 1×2 components, whereas the second group has one more pulse generator cell, i.e. 1×3 components.

For our experiments, we use the property patterns provided in Table 3. Table 4 shows the verification results for the properties given in Table 3 using two different groups. NUSMV has returned TRUE for all the properties. In the group with 1×2 components, we have verified that the sender cell (`sender1`) can produce a signalling molecule and transmit it to the pulsing cell (`pulsing1`). In addition, the pulse generator cell can use that signal to produce the green florescent protein (`proteinGFP`). In the group with 1×3 components, we have verified similar properties. In addition, we have verified that the first pulsing cell (`pulsing1`) can transmit the signalling molecule to the second pulsing cell (`pulsing2`).

Reduced Rule Sets. Using a larger sets of components, we want to prove that the signalling molecules can be transmitted to the furthest pulsing cells. However, when we increase the number of cells, verification becomes no longer feasible due to the state explosion problem. In order to achieve the verification results within a reasonable time, we have compacted the rules sets such that an entire chain of reactions is replaced by a fewer simple rules. Consequently, the overall number of interactions is reduced and all the species which do not appear in the new set of rules are removed from the model. These changes are made in the non-deterministic models as these are used for qualitative analyses where the concentration of certain molecules is not significant or chain of reaction already analysed can be replaced by some abstractions mimicking their behaviour through simpler rewriting mechanisms.

Here, we define a group of cells with 1×5 components, where 1 sender and 4 pulsing cells are placed in row. For this scenario, we could verify the same properties in Table 4 using the reduced rule sets (as defined in Section 4.3). In addition, we have verified additional properties to analyse

Table 4: Verification experiments for the complete rule sets.

Lattice	Property	X, Y
1 × 2	Prop. 1	X = sender ₁ .signal30C6 X = pulsing ₁ .signal30C6
	Prop. 2	X = pulsing ₁ .signal30C6, Y = pulsing ₁ .proteinGFP X = sender ₁ .signal30C6, Y = pulsing ₁ .proteinGFP
	Prop. 3	X = pulsing ₁ .signal30C6, Y = pulsing ₁ .proteinGFP X = sender ₁ .signal30C6, Y = pulsing ₁ .proteinGFP
1 × 3	Prop. 1	X = pulsing ₂ .signal30C6 X = pulsing ₂ .proteinGFP
	Prop. 2	X = pulsing ₁ .signal30C6, Y = pulsing ₂ .proteinGFP X = sender ₁ .signal30C6, Y = pulsing ₂ .proteinGFP
	Prop. 3	X = pulsing ₁ .signal30C6, Y = pulsing ₂ .signal30C6 X = sender ₁ .signal30C6, Y = pulsing ₂ .signal30C6

Table 5: Verification experiments for the reduced rule sets.

Lattice	Property	X, Y
1 × 5	Prop. 1	X = pulsing ₄ .signal30C6 X = pulsing ₄ .proteinGFP
	Prop. 2	X = pulsing ₁ .signal30C6, Y = pulsing ₄ .proteinGFP X = sender ₁ .signal30C6, Y = pulsing ₄ .proteinGFP
	Prop. 3	X = pulsing ₃ .signal30C6, Y = pulsing ₄ .signal30C6 X = pulsing ₃ .signal30C6, Y = pulsing ₄ .proteinGFP

the other pulsing cells. Table 5 shows these properties, for which NUSMV has returned TRUE. The verification results show that the sender cell can produce the signalling molecule and transmit it to the adjacent pulsing cell, and the pulsing cells can use the signalling molecule to produce `proteinGFP` and transmit it to the its neighbour pulsing cells.

6 Conclusions

In this paper, we have presented two extensions to KPWORKBENCH: a formal verification tool based on the NUSMV model checker and a large scale simulation environment using FLAME, a platform for agent-based modelling on parallel architectures. The use of these two tools for modelling and analysis of biological systems is illustrated with a pulse generator, a synthetic biology system. We have provided both the stochastic model as stochastic P systems and the non-deterministic model as kernel P systems as well as a reduced model. We have also provided both simulation and verification results, confirming the desired behaviour of the pulse generator system.

The NUSMV tool currently works for a restricted subset of the kP-Lingua language, and we only consider one execution strategy, the nondeterministic choice. As a future work, we will extend the compatibility of the tool to cover the full language and the other execution strategies, e.g. sequence and maximal parallelism. The future work will also involve modeling, simulation and verification of even more complex biological systems as well as performance comparisons of simulators and model checking tools integrated within KPWORKBENCH.

Acknowledgements. The work of FI, LM and IN was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688). SK acknowledges the support provided for synthetic biology research by EPSRC

ROADBLOCK (project number: EP/I031812/1). MB is supported by a PhD studentship provided by the Turkey Ministry of Education.

The authors would like to thank Marian Gheorghe for his valuable comments to this paper.

References

1. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipaté, F.: High performance simulations of kernel P systems. In: The 16th IEEE International Conference on High Performance Computing and Communications (2014)
2. Basu, S., Mehreja, R., Thiberge, S., Chen, M.T., Weiss, R.: Spatio-temporal control of gene expression with pulse-generating networks. *PNAS* 101(17), 6355–6360 (2004)
3. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F.J., Krasnogor, N., Gheorghe, M.: Infobiotics Workbench: A P systems based tool for systems and synthetic biology. In: [7], pp. 1–41. Springer (2014)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An open source tool for symbolic model checking. In: Proc. International Conference on Computer-Aided Verification (CAV 2002). LNCS, vol. 2404. Springer, Copenhagen, Denmark (July 2002)
5. Dragomir, C., Ipaté, F., Konur, S., Lefticaru, R., Mierlă, L.: Model checking kernel P systems. In: 14th International Conference on Membrane Computing. LNCS, vol. 8340, pp. 151–172. Springer (2013)
6. FLAME: Flexible large-scale agent modeling environment (available at <http://www.flame.ac.uk/>)
7. Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.): Applications of Membrane Computing in Systems and Synthetic Biology. Springer (2014)
8. Gheorghe, M., Ipaté, F., Dragomir, C., Mierlă, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P systems - Version 1. 12th BWMC pp. 97–124 (2013)
9. Gillespie, D.: A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics* 22(4), 403–434 (1976)
10. Holzmann, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering* 23(5), 275–295 (1997)
11. Ipaté, F., Bălănescu, T., Kefalas, P., Holcombe, M., Eleftherakis, G.: A new model of communicating stream X-machine systems. *Romanian Journal of Information Science and Technology* 6, 165–184 (2003)
12. Konur, S., Gheorghe, M., Dragomir, C., Ipaté, F., Krasnogor, N.: Conventional verification for unconventional computing: a genetic XOR gate example. *Fundamenta Informaticae* (2014)
13. Konur, S., Gheorghe, M., Dragomir, C., Mierlă, L., Ipaté, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. *ACS Synthetic Biology* (2014)
14. KPWorkbench (available at <http://kpworkbench.org>)
15. Niculescu, I.M., Gheorghe, M., Ipaté, F., Stefanescu, A.: From kernel P systems to X-machines and FLAME. *Journal of Automata, Languages and Combinatorics* To appear (2014)
16. Păun, G.: Computing with membranes. *Journal of Computer and System Sciences* 61(1), 108–143 (2000)
17. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
18. Romero-Campero, F.J., Twycross, J., Cao, H., Blakes, J., Krasnogor, N.: A multiscale modeling framework based on P systems. In: Membrane Computing, LNCS, vol. 5391, pp. 63–77. Springer (2009)
19. Sanassy, D., Fellermann, H., Krasnogor, N., Konur, S., Mierlă, L., Gheorghe, M., Ladroue, C., Kalvala, S.: Modelling and stochastic simulation of synthetic biological boolean gates. In: The 16th IEEE International Conference on High Performance Computing and Communications (2014)

Appendix

Algorithm 1 Transforming a kP Systems into Flame algorithm

```

1: procedure ADDTTRANSITION(startState, stopState, strategy, guard)
  ▷ procedure adding the appropriate transition strategy to the current agent stack given as parameter
  and FLAME function applying rules conforming to execution strategy
  ▷ guard is an optional parameter that represents the transition guard
2:   if strategy is Sequence then
3:     agentTtransitions.Push(startState, stopState, SequenceFunction, guard)
     ▷ FLAME function SequenceFunction applies rules in sequentially mode
4:   else if strategy is Choice then
5:     agentTtransitions.Push(startState, stopState, ChoiceFunction, guard)
     ▷ FLAME function ChoiceFunction applies rules in choice mode
6:   else if strategy is ArbitraryParallel then
7:     agentTtransitions.Push(startState, stopState, ArbitraryParallelFunction, guard)
     ▷ FLAME function ArbitraryParallelFunction applies rules in arbitrary parallel mode
8:   else if strategy is MaximalParallel then
9:     agentTtransitions.Push(startState, stopState, MaximalParallelFunction, guard)
     ▷ FLAME function MaximalParallelFunction applies rules in maximal parallel mode
10:  end if
11: end procedure
12:
  ▷ main algorithm for traforming a kP system into Flame
13:
14: agentsStates.Clear()
15: agentsTtransitions.Clear()
  ▷ empty state and transition stacks of agents
16: foreach membrane in kPSystem do
  ▷ for each membrane of kP system build corresponding agent, consisting of states and transitions
17:   agentStates.Clear()
18:   agentTtransitions.Clear()
  ▷ empty state and transition stacks of agent that is built for the current membrane
19:   agentStates.Push(startState)
  ▷ adding the initial state of the X machine
20:   agentStates.Push(initializationState)
  ▷ adding initialization state
21:   agentTtransitions.Push(startState, initializationState)
  ▷ adding transition between the initial and initialization states; this transition performs objects
  allocation on rules and other initializations
22:   foreach strategy in membrane do
  ▷ for each strategy of the current membrane the corresponding states and transitions are built
23:     previousState = agentStates.Top()
     ▷ the last state is stored in a temporary variable
24:     if is first strategy and strategy.hasNext() then
     ▷ when the strategy is the first of several, state and transition corresponding to the execution
     strategy are added
25:       agentStates.Push(strategy.Name)
26:       AddTtransition(previousState, strategy.Name, strategy)
27:     else
28:       if not strategy.hasNext() then
     ▷ if it is the last strategy, the transition corresponding to the execution strategy is added
29:         AddTtransition(previousState, applyChangesState, strategy)
30:       else

```

Algorithm 1 Transforming a kP Systems into Flame algorithm (continued)

```

31:     agentStates.Push(strategy.Name)
    ▷ add corresponding state of the current strategy
32:     if strategy.Previous() is Sequence then
    ▷ verify that previous strategy is of sequence type
33:         AddTtransition(previousState, strategy.Name, strategy, IsApplyAllRules)
        ▷ add transition from preceding strategy state to the current strategy state. The guard
        is active if all the rules have been applied in the previous strategy transition.
34:         agentTtransitions.Push(previousState, applyChangesState, IsNotApplyAllRules)
        ▷ add transition from preceding strategy state to state where changes produced by rules
        are applied. The guard is active if not all rules have been applied in the previous
        strategy transition
35:     else
36:         AddTtransition(previousState, strategy.Name, strategy)
        ▷ add transition from preceding strategy state to the current strategy state
37:         agentTtransitions.Push(previousState, applyChangesState, IsApplyStructureRule)
        ▷ add transition from preceding state strategy to state in which changes produced by the
        applied rules are committed. The guard is active when the structural rule has been
        applied on the previous strategy transition
38:     end if
39: end if
40: end if
41: end for
42: agentStates.Push(applyChangesState)
    ▷ adding state in which changes produced by the applied rules are committed
43: agentTtransitions.Push(applyChangesState, receiveState)
    ▷ adding transition on which changes produced by the applied rules are committed
44: agentStates.Push(receiveState)
    ▷ add state that receives objects sent by applying the communication rules in other membranes
45: agentTtransitions.Push(receiveState, s0State)
    ▷ add transition that receives objects sent by applying the communication rules in other membranes
46: agentStates.Push(s0State)
    ▷ add an intermediary state
47: agentTtransitions.Push(s0State, endState, IsNotApplyStructureRule)
    ▷ add transition to the final state in which nothing happens unless a structural rule was applied
48: agentTtransitions.Push(s0State, endState, IsApplyStructureRule)
    ▷ add the transition to the final state on which structural changes are made if the structure rule has
    been applied
49: agentStates.Push(endState)
    ▷ add the final state
50: agentsStates.PushAll(agentStates.Content())
    ▷ add the contents of the stack that holds the current agent states to the stack that holds the states
    of all agents
51: agentsTtransitions.PushAll(agentStates.Content())
    ▷ add the contents of the stack that holds the current agent transitions to the stack that holds the
    transitions of all agents
52: end for

```

Table 6: Multiset rules ($R_{pulsing}$) of the SP systems model of the pulsing cell.

Rule	Kinetic constant
r_1 : PluxL_geneLuxR $\xrightarrow{k_1}$ PluxL_geneLuxR + rnaLuxR_RNAP	$k_1 = 0.1$
r_2 : rnaLuxR_RNAP $\xrightarrow{k_2}$ rnaLuxR	$k_2 = 3.2$
r_3 : rnaLuxR $\xrightarrow{k_3}$ rnaLuxR + proteinLuxR_Rib	$k_3 = 0.3$
r_4 : rnaLuxR $\xrightarrow{k_4}$	$k_4 = 0.04$
r_5 : proteinLuxR_Rib $\xrightarrow{k_5}$ proteinLuxR	$k_5 = 3.6$
r_6 : proteinLuxR $\xrightarrow{k_6}$	$k_6 = 0.075$
r_7 : proteinLuxR + signal30C6 $\xrightarrow{k_7}$ proteinLuxR_30C6	$k_7 = 1.0$
r_8 : proteinLuxR_30C6 $\xrightarrow{k_8}$	$k_8 = 0.0154$
r_9 : proteinLuxR_30C6 + proteinLuxR_30C6 $\xrightarrow{k_9}$ LuxR2	$k_9 = 1.0$
r_{10} : LuxR2 $\xrightarrow{k_{10}}$	$k_{10} = 0.0154$
r_{11} : LuxR2 + PluxR_geneCI $\xrightarrow{k_{11}}$ PluxR_LuxR2_geneCI	$k_{11} = 1.0$
r_{12} : PluxR_LuxR2_geneCI $\xrightarrow{k_{12}}$ LuxR2 + PluxR_geneCI	$k_{12} = 1.0$
r_{13} : PluxR_LuxR2_geneCI $\xrightarrow{k_{13}}$ PluxR_LuxR2_geneCI + rnaCI_RNAP	$k_{13} = 1.4$
r_{14} : rnaCI_RNAP $\xrightarrow{k_{14}}$ rnaCI	$k_{14} = 3.2$
r_{15} : rnaCI $\xrightarrow{k_{15}}$ rnaCI + proteinCI_Rib	$k_{15} = 0.3$
r_{16} : rnaCI $\xrightarrow{k_{16}}$	$k_{16} = 0.04$
r_{17} : proteinCI_Rib $\xrightarrow{k_{17}}$ proteinCI	$k_{17} = 3.6$
r_{18} : proteinCI $\xrightarrow{k_{18}}$	$k_{18} = 0.075$
r_{19} : proteinCI + proteinCI $\xrightarrow{k_{19}}$ CI2	$k_{19} = 1.0$
r_{20} : CI2 $\xrightarrow{k_{20}}$	$k_{20} = 0.00554$
r_{21} : LuxR2 + PluxPR_geneGFP $\xrightarrow{k_{21}}$ PluxPR_LuxR2_geneGFP	$k_{21} = 1.0$
r_{22} : PluxPR_LuxR2_geneGFP $\xrightarrow{k_{22}}$ LuxR2 + PluxPR_geneGFP	$k_{22} = 1.0$
r_{23} : LuxR2 + PluxPR_CI2_geneGFP $\xrightarrow{k_{23}}$ PluxPR_LuxR2_CI2_geneGFP	$k_{23} = 1.0$
r_{24} : PluxPR_LuxR2_CI2_geneGFP $\xrightarrow{k_{24}}$ LuxR2 + PluxPR_CI2_geneGFP	$k_{24} = 1.0$
r_{25} : CI2 + PluxPR_geneGFP $\xrightarrow{k_{25}}$ PluxPR_CI2_geneGFP	$k_{25} = 5.0$
r_{26} : PluxPR_CI2_geneGFP $\xrightarrow{k_{26}}$ CI2 + PluxPR_geneGFP	$k_{26} = 0.0000001$
r_{27} : CI2 + PluxPR_LuxR2_geneGFP $\xrightarrow{k_{27}}$ PluxPR_LuxR2_CI2_geneGFP	$k_{27} = 5.0$
r_{28} : PluxPR_LuxR2_CI2_geneGFP $\xrightarrow{k_{28}}$ CI2 + PluxPR_LuxR2_geneGFP	$k_{28} = 0.0000001$
r_{29} : PluxPR_LuxR2_geneGFP $\xrightarrow{k_{29}}$ PluxPR_LuxR2_geneGFP + rnaGFP_RNAP	$k_{29} = 4.0$
r_{30} : rnaGFP_RNAP $\xrightarrow{k_{30}}$ rnaGFP	$k_{30} = 3.36$
r_{31} : rnaGFP $\xrightarrow{k_{31}}$ rnaX + proteinGFP_Rib	$k_{31} = 0.667$
r_{32} : rnaGFP $\xrightarrow{k_{32}}$	$k_{32} = 0.04$
r_{33} : proteinGFP_Rib $\xrightarrow{k_{33}}$ proteinGFP	$k_{33} = 3.78$
r_{34} : proteinGFP $\xrightarrow{k_{34}}$	$k_{34} = 0.0667$

 Table 7: Multiset rules ($R''_{pulsing}$) of the kP systems model of the pulsing cell.

Rule
r_1 : PluxL_geneLuxR \rightarrow PluxL_geneLuxR + rnaLuxR_RNAP
r_2 : proteinLuxR \rightarrow
r_3 : proteinLuxR + signal30C6 \rightarrow proteinLuxR_30C6
r_4 : proteinLuxR_30C6 \rightarrow
r_5 : proteinLuxR_30C6 + PluxPR_geneGFP \rightarrow PluxPR_LuxR2_geneGFP
r_6 : PluxPR_LuxR2_geneGFP \rightarrow PluxPR_LuxR2_geneGFP + proteinGFP
r_7 : proteinGFP \rightarrow
r_8 : signal30C6 \rightarrow signal30C6 (pulsing)

High Performance Simulations of Kernel P Systems

Mehmet E. Bakir, Savas Konur, Marian Gheorghe
Department of Computer Science
University of Sheffield
Sheffield, UK
Email: mebakir1@sheffield.ac.uk

Ionut Niculescu, Florentin Ipatu
Department of Computer Science
University of Bucharest
Bucharest, Romania
Email: ionutmihainiculescu@gmail.com

Abstract—The paper presents the use of a membrane computing model for specifying a synthetic biology pulse generator example and discusses some simulation results produced by the tools associated with this model and compare their performances. The results show the potential of the simulation approach over the other analysis tools like model checkers.

I. INTRODUCTION

Nature inspired computing, including biological and chemical (or molecular) computing, has become a very intensively investigated research area, with a consistent body of theoretical research, and with many interesting applications and challenging problems [1]. One of the most recently introduced area of natural computing is *membrane computing*. This has been conceived as a computational paradigm inspired by the structure and behaviour of the living cell [2]. Many models have been considered and studied, and substantial theoretical results related to the computational power and complexity aspects have been obtained [3]. These models are called *membrane systems* or *P systems*. Interesting applications in systems and synthetic biology have been provided, using a set of methods and tools based on this computational model [4]. In the last years there have been attempts to create more general membrane computing models, which allow the specification of various classes of problems defined with different membrane computing models and providing mechanisms to analyse such systems and simulate their behaviour. One such model, called *kernel P systems* [5], [6], has been recently introduced and some tools for the simulation and verification of the systems specified with this formalism have been built.

In this paper, we present the use of kernel P systems for specifying a synthetic biology pulse generator example and discuss some simulation results produced by the tools associated with this model and compare their performances. The results show the potential of the simulation approach over the other analysis tools like model checkers, which heavily suffer from the well-known *state explosion* problem.

In Section II, we very briefly describe kernel P systems. In Section III, we present the simulation frameworks, supporting the simulation of kernel P system models. Section IV describes the synthetic pulse generator, and presents the experimental results. Section V draws some conclusions and provides some future research directions.

II. KERNEL P SYSTEMS

Kernel P systems (kP systems for short) are multiset transformation mechanisms consisting of *compartments* linked

by some communication channels; each compartment contains *multisets of objects* and *rewriting and communication rules* which transform the multisets of objects and send them to neighbour compartments. The system evolves in steps and at each step, in each compartment the rules are applied in accordance with a certain execution strategy. For the example considered in this paper, in each compartment a rule is executed per step, non-deterministically chosen from those applicable at that moment. The system starts having in each compartment some initial multiset of objects. A formal definition of these models is available from [5], [6].

Kernel P systems are supported by a software framework, called KPWORKBENCH [7], [8], which integrates a set of tools enabling simulation and model checking of kP systems (e.g. genetic Boolean gates [9]). The KPWORKBENCH tool implements several translations that connect several target specifications employed for kP system models. In [7], the model checker SPIN is utilised in order to verify properties of the kP system models. The KPWORKBENCH also consists of a native simulator which allows the execution of the models written with the kP system formalism. Recently a translator of kP system models to FLAME has been produced, based on a method that allows the expression of kP systems as a set of communicating X-machines [10].

III. SIMULATION FRAMEWORKS FOR KP SYSTEMS

In [11] we have shown the benefits of using a model checker for verifying various properties of the system and for checking the validity of the model. However, we face the well-known problem of state explosion for such approaches and we can only model check very simple systems with very few compartments. The simulation allows us to look at much larger systems and check various results, either final or intermediary ones.

Kernel P system models are specified and represented by a simple and intuitive modelling language, called *kP-Lingua* [12]. kP-Lingua provides for a kP system model a representation into a machine readable format. It also has its own syntax and specific ways of creating compartment types, their instances and connections between them.

KPWORKBENCH integrates a simulation tool, KPWORKBENCH SIMULATOR, and provides mechanisms to translate kP-Lingua specifications to FLAME.

A. KPWORKBENCH SIMULATOR

KPWORKBENCH SIMULATOR is a custom simulation tool, implemented in C# programming language. The tool requires a

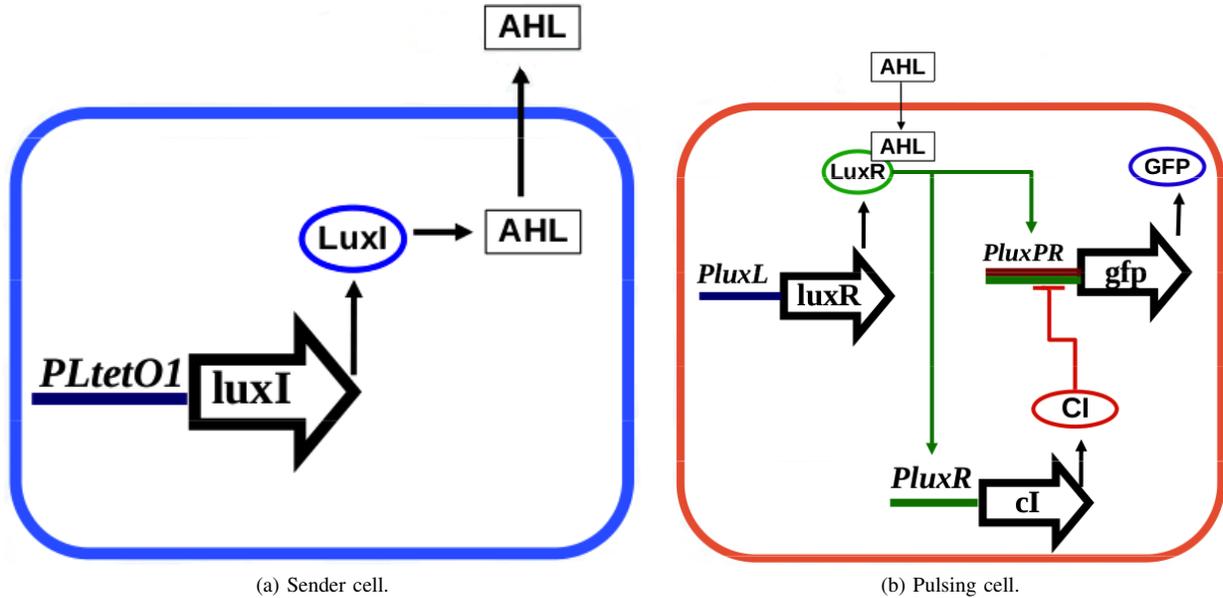


Fig. 1: Two cell types of the pulse generator system (taken from [11]).

kP system model specified in kP-Lingua as input and provides traces of execution for a kP system model. This is translated into an internal data structure, which allows to represent compartments, containing multisets of objects and rules, and their connections with other compartments. The execution strategy in each compartment is interpreted step by step. The simulator provides a command line user interface displaying the current configuration (the content of each compartment) at each step. The output can be printed on command prompt or can be redirected to a file. Depending on the starting step and the granularity of the output, the amount of the printing data will change, which can significantly affect the execution time. The tool is particularly useful for quickly running sanity check on a kP system model, for checking the temporal evolution of the system and for inferring useful information from the simulation results.

B. FLAME

FLAME [13] is a general purpose agent based framework, built on top of the X-machine formalism, a state based model with transformation functions associated to the transitions of the model. It represents the structure of the state machine using an XML format and the transformation functions in standard C. FLAME has become very popular and widely used for numerous applications. The latest developments of FLAME have been focussing on developing variants for high performance computers [13].

The current translator from kP-Lingua maps kP systems into FLAME agents with internal behaviour consisting of rule rewriting and communication. The FLAME environment then executes the model the requested number of steps, storing intermediary results that can be afterwards analysed or interpreted.

A kP system is transformed into a communicating X-machine system by constructing, for each membrane, a com-

municating X-machine [10] that simulates its behaviour. An additional X-machine, that helps the synchronization of the others, is also built. Each execution strategy of the membrane corresponds to a transition in the communicating X-machine. In FLAME, the communicating X-machines are transformed into agents. Here, the additional X-machine is no longer needed since the synchronization is achieved through message passing.

IV. PERFORMANCE COMPARISON

In this section, we will evaluate the performances of two simulators, integrated into the KPWORKBENCH platform. The performances are very similar in small models. Thus, the evaluation should be performed in large systems. We therefore choose a model from synthetic biology, because synthetic biology models can be very large and it will be a good test case for the simulators. Here, we choose the synthetic *pulse generator*.

A. Pulse Generator

The *pulse generator* [14] is a synthetically constructed colony of bacteria, containing two types of cells: *sender* and *pulsing* (see Figure 1). The sender cells synthesise a signalling protein, transmitted through the pulsing cells. The pulsing cells express the green fluorescent protein (GFP) triggered by the signalling molecules, and propagate the excess signalling molecules to the neighbouring cells. The biological process illustrated in Figure 1 can be summarised as follows [11]:

“Sender cells contain the gene *luxI* from *Vibrio fischeri*. This gene codifies the enzyme *LuxI* responsible for the synthesis of the molecular signal 3OC12HSL (AHL). The *luxI* gene is expressed constitutively under the regulation of the promoter

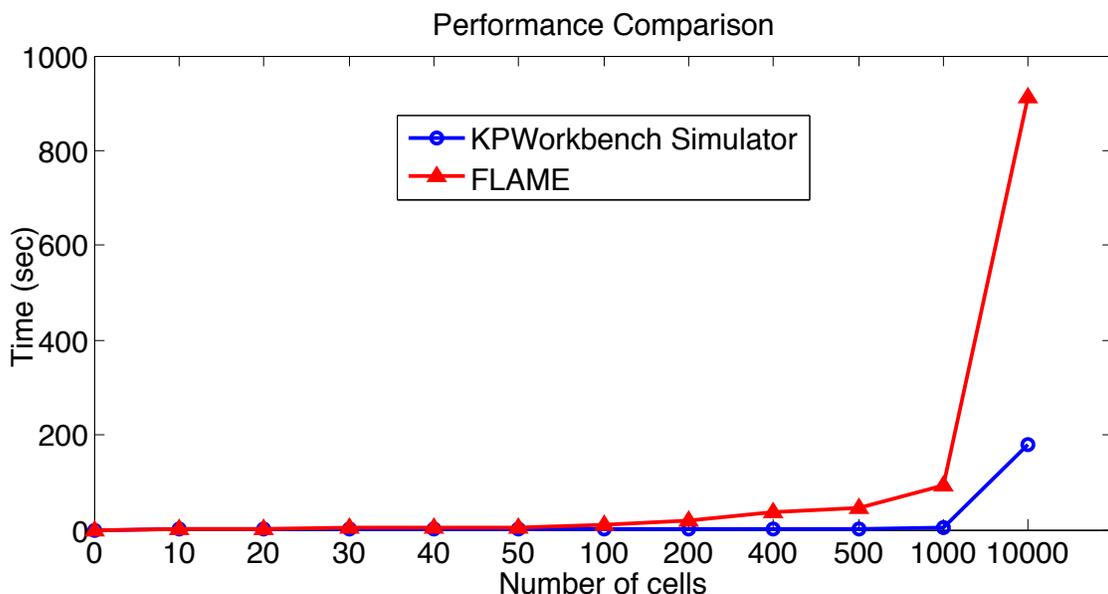


Fig. 2: The comparative simulation results for KPWORKBENCH and FLAME

PLtetO1 from the tetracycline resistance transposon.”

“Pulsing cells contain the *luxR* gene from *Vibrio fischeri* that codifies the 3OC12HSL receptor protein LuxR. This gene is under the constitutive expression of the promoter *PluxL*. It also contains the gene *cI* from lambda phage codifying the repressor CI under the regulation of the promoter *PluxR* that is activated upon binding of the transcription factor LuxR_3OC12. Finally, this bacterial strain carries the gene *gfp* that codifies the green fluorescent protein under the regulation of the synthetic promoter *PluxPR* combining the *Plux* promoter (activated by the transcription factor LuxR_3OC12) and the PR promoter from lambda phage (repressed by the transcription factor CI).”

The bacterial strains above are distributed in a specific spatial distribution as a lattice with n rows and m columns. The first two columns consist of sender cells, whereas the rest are pulsing cells. The behaviour of each sender cell is described by 7 rewriting and 1 communication rules and that of the pulsing cell by 34 rewriting and 1 communication rules. The entire model is described in [11] where different properties of the system, both quantitative and qualitative, are verified for small size lattices. Here we consider these two cell types, with the above mentioned rules, but with various lattices which are described in the next section.

The pulse generator is a challenging example, as it is compartmental by design and the dynamic behaviour of each bacterial strain is governed by a large number of kinetic rules. When the number of compartments are increased, the size of the model grows very sharply and the execution of simulations becomes demanding. This makes the pulse generator a good test case for our simulators.

B. Experiments

We will consider a simpler lattice with only a sender and a pulsing cell, but this system will be multiplied by 10, 20, 30, 40, 50, 100, 200, 400, 500, 1000, and 10000 times. For the purpose of these experiments these systems are equivalent to lattices with sizes in the range 10 .. 10000, which are significantly more complex than those described in [11]. Each case will be executed 5 times and the average time calculated. These will be executed with the native KPWORKBENCH simulator and with the FLAME simulator. We note that the system model is described as a kP system model, which is automatically translated into the FLAME simulator. Also, the KPWORKBENCH simulator accepts kP system models, as input.

The results of the simulations on both the KPWORKBENCH simulator and FLAME are comparatively presented in Figure 2. The x axis gives the number of send-pulse pairs of cells, while the y axis indicates the time in seconds. The experiments were performed on a PC with the following configuration: Intel(R) Core(TM)2 Quad CPU - Q6600 2,4Ghz, 4GB RAM. In what follows we explain the performance difference between the two simulators.

In the KPWORKBENCH SIMULATOR, each membrane of the kP system is represented by an instance of a class, transformed from the kP-Lingua language. This approach makes the simulation to be performed in a single memory space, that scales according to the number of membranes used in the model and the number of objects resulting from applying the rules in each simulation step.

In FLAME each agent is represented by an acyclic X-machine (no loops are allowed in order to ensure the completion of the execution of the agent). The agent is executed by passing from one state to another in the X-machine and processing data using functions that are attached to the transitions. When the X-machine reaches the final state, the data

is written to the hard disk and it is then used as input for the next iteration. An important characteristic of FLAME is that it first reads the input data, stored in XML format files, from the hard drive and writes it back at the end of each iteration.

In FLAME, each membrane of the kP system is represented by an agent. The rules are stored together with the membrane multiset as agent data. For each type of membrane from the kP system, a type of agent is defined, and for each execution strategy of the membrane, states are created in the X-machine. Transitions between the two states are represented by C functions that are executed in FLAME when passing from one state to another. Each type of strategy defines a specific function that applies the rules according to the execution strategy.

Given the way the agents for simulating a kP system in FLAME are defined, the volume of data increases with the number of types of membranes, the number of their instances and the size of their multisets. (Note that, since there are no structural rules in our model, the number and structure of membranes remain unchanged throughout the simulation, so the execution time will depend only on the size of the processed multisets.) Consequently, the more data we have, the more time for reading and writing data from or to the hard disk is required. This explains the higher execution time in the case of the FLAME simulator than for the KPWORKBENCH SIMULATOR. It is expected that at least for systems of the same type, i.e., using one single rule per step, the behaviour of the two platforms will be similar. A better correlation between the type of the system, the number of compartments and number of rules, and the behaviour of these platforms will be investigated in a forthcoming paper.

On the other hand, the distributed architecture of Flame allows the simulation to be run on parallel supercomputers with great performance improvements [13]. (For the moment, the implementation of kP Workbench is not suitable for running on parallel computers, but this issue may be considered at a later stage.) Significant performance gains could also be obtained by using solid-state drives (SSDs) for data storage, with lower access time than traditional HDDs, but this is beyond the scope of this paper.

V. CONCLUSION

In this paper we have presented the simulation results of a synthetic biology model coded as a kernel P system, a nature inspired computational paradigm, executed under two different simulation environments. The results presented show the capability of the simulation environments to deal with large scale models - up to 10000 components, each with more than 30 transitions - and consequently with the benefits of a complementary approach to model verification methods, already used for such systems.

The experiments performed show some expected behaviour, whereby a specialised simulation tool, KPWORKBENCH SIMULATOR, provides better results, in terms of execution time, than a general purpose simulation environment, namely FLAME. Both tools produce the same behaviour starting from the same specification, kP-Lingua description, and the translation process is obtained in an automatic way.

In the long term, we aim to show the way the results of the simulation and those of formal verification complement each other for a better understanding of the system behaviour.

ACKNOWLEDGMENT

IN, FI and MG are supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688). SK and MG acknowledge the support provided for the synthetic biology research by EPSRC ROADBLOCK (project number: EP/I031812/1). MB is supported by a PhD studentship provided by the Turkey Ministry of Education.

REFERENCES

- [1] G. Rozenberg, T. Bäck, and J. N. Kok, Eds., *Handbook of Natural Computing*. Springer, 2012.
- [2] G. Păun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000.
- [3] G. Păun, G. Rozenberg, and A. Salomaa, Eds., *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
- [4] P. Frisco, M. Gheorghe, and M. J. Pérez-Jiménez, Eds., *Applications of Membrane Computing in Systems and Synthetic Biology*. Springer, 2014.
- [5] M. Gheorghe, F. Ipate, and C. Dragomir, "Kernel P systems," pp. 153–170, 2012.
- [6] M. Gheorghe, F. Ipate, C. Dragomir, L. Mierlă, L. Valencia-Cabrera, M. García-Quismondo, and M. J. Pérez-Jiménez, "Kernel P systems - Version 1," pp. 97–124, 2013.
- [7] C. Dragomir, F. Ipate, S. Konur, R. Lefticaru, and L. Mierlă, "Model checking kernel P systems," in *Proc. 14th Conference on Membrane Computing*, ser. LNCS, vol. 8340. Springer, 2014, pp. 151–172.
- [8] M. E. Bakir, F. Ipate, S. Konur, L. Mierlă, and I. Niculescu, "Extended simulation and verification platform for kernel p systems," in *15th International Conference on Membrane Computing*, 2014, p. To Appear.
- [9] S. Konur, M. Gheorghe, C. Dragomir, F. Ipate, and N. Krasnogor, "Conventional verification for unconventional computing: a genetic XOR gate example," *Fundamenta Informaticae*, p. To Appear, 2014.
- [10] I. Niculescu, F. I. M. Gheorghe, and A. Stefanescu, "From kernel P systems to X-machines and FLAME," *Journal of Automata, Languages and Combinatorics*, to appear.
- [11] J. Blakes, J. Twycross, S. Konur, F. Romero-Campero, N. Krasnogor, and M. Gheorghe, "Infobiotics workbench: A P systems based tool for systems and synthetic biology," in *Applications of Membrane Computing in Systems and Synthetic Biology*, ser. Emergence, Complexity and Computation. Springer International Publishing, 2014, vol. 7, pp. 1–41.
- [12] M. Gheorghe, F. Ipate, C. Dragomir, L. Mierla, L. Valencia-Cabrera, M. García-Quismondo, and M. J. Pérez-Jiménez, "Kernel P systems," *Eleventh Brainstorming Week on Membrane Computing*, pp. 97–124, 2013.
- [13] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough, "Exploitation of high performance computing in the flame agent-based simulation framework," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICES)*, 2012 IEEE 14th International Conference on, June 2012, pp. 538–545.
- [14] S. Basu, R. Mehreja, S. Thiberge, M.-T. Chen, and R. Weiss, "Spatiotemporal control of gene expression with pulse-generating networks," *PNAS*, vol. 101, no. 17, pp. 6355–6360, 2004.

Chapter 6

Comparative Analysis of Statistical Model Checking Tools

In the previous chapter, we demonstrated how biological systems can be expressed by using computational models and how they can be analysed with simulation and model checking techniques. Model checking can be applied exhaustively or approximately. The exact model checking method verifies a property by investigating the whole state space. As a result, it can suffer from the state explosion problem, and in practice, it can be exercised only on small models. Alternatively, statistical model checking considers a fraction of simulation traces rather than all states and can provide only approximate correctness of a queried property. Typically, statistical model checkers can verify larger models, and they are faster than exact model checkers. The advantages of statistical model checking have persuaded scientists to develop a number of tools to embody this formalism. Although the diversity of the tools gives flexibility to users, it comes with a cost of lack of clarity as to which statistical model checking tool is best for a given model. The study presented in this chapter reviewed some of the popular statistical model checking tools, and it is the third paper included in this thesis.

Third paper: Mehmet Emin Bakir, Marian Gheorghe, Savas Konur, and Mike Stannett. Comparative analysis of statistical model checking tools. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing: 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers*, pages 119–135. Springer International Publishing, Cham, 2017.

The study introduces five statistical model checking tools, compares their modelling and specification languages, the property patterns that they support, and their ease of use. It presents the performance benchmarking of the tools on verification of 465 biological models against five property patterns.

We developed a custom tool for translating the biological models to different model checkers’ modelling and specification languages and recording the verification time. The application is later extended to include more components. The final source code is available on www.github.com/meminbakir/smcp. The initial components developed and used for this study are: ‘mce component (1929 *lines of code* (LOC))’ that manages overall performance benchmarking. ‘modify component (624 LOC)’ modifies the biological model to be ready for translation, for example, it fixes the stochastic rates of reactions. ‘mchecking component (7988 LOC)’ translates biological models to model checkers’ specification and performs verification. ‘output component (183 LOC)’ is responsible for printing out the messages to the users. ¹.

The overall performance benchmarking was very time consuming, we run each tool three times for the verification each model and the property pattern. Each verification time is limited to one hour. The worst-case scenario of total verification time is, 1 hour*(5 SMC X 465 models X 5 property pattern X 3 execution). However, fortunately, small models verified quite faster, therefore with our computer settings (details are in the Experimental Findings section), the performance benchmarking process is completed within 3-4 months, though still a significant amount of time.

The experiments showed that the performance of different tools significantly changes per property pattern type, and indeed, per model size. The study concludes that in

¹Lines of code is calculated with “`find . -name ‘*’ | xargs wc -l`” MAC command.

some cases the best model checker can be identified by considering only the model size and the property pattern. Therefore, this study provides guidance for specific scenarios. For example, when the model size is very small or large, users can quickly decide which model checker is going to be the fastest tool for their experiments, by just examining their model size, and the pattern of the property they want to verify. The study also maps borders where the best choice is less clear-cut. For such cases, examining the model size and property pattern parameters alone is not enough. Therefore, we should explore more model features for determining the fastest model checker. The findings point out to a clear need for automating the best SMC tool selection process.

Comparative Analysis of Statistical Model Checking Tools

Mehmet Emin Bakir¹, Marian Gheorghe², Savas Konur², and Mike Stannett¹

¹ Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello, Sheffield, S1 4DP, UK
mebakir1@sheffield.ac.uk, m.stannett@sheffield.ac.uk

² School of Electrical Engineering and Computer Science, University of Bradford
West Yorkshire, Bradford, BD7 1DP, UK
m.gheorghe@bradford.ac.uk, s.konur@bradford.ac.uk

Abstract. Statistical model checking is a powerful and flexible approach for formal verification of computational models, e.g. P systems, which can have very large search spaces. Various statistical model checking tools have been developed, but choosing the most efficient and appropriate tool requires a significant degree of experience, not only because different tools have different modelling and property specification languages, but also because they may be designed to support only a certain subset of property types. Furthermore, their performance can vary depending on the property types and membrane systems being verified. In this paper, we evaluate the performance of various common statistical model checkers based on a pool of biological models. Our aim is to help users select the most suitable SMC tools from among the available options, by comparing their modelling and property specification languages, capabilities and performances.

Keywords: membrane computing, P systems, statistical model checking, biological models, performance benchmarking

1 Introduction

In order to understand the structure and functionality of biological systems, we need methods which can highlight the spatial and time-dependent evolution of systems. To this end, researchers have started to utilize the computational power of machine-executable models, including implementations of membrane system models, to get a better and deeper understanding of the spatial and temporal features of biological systems [21]. In particular, the executable nature of computational models enables scientists to conduct experiments, *in silico*, in a fast and cheap manner.

The vast majority of models used for describing biological systems are based on ordinary differential equations (ODEs) [9], but researchers have recently started to use *computational models* as an alternative to mathematical modelling. The basis of such models is *state machines*, which can be used to model

numerous variables and relate different system states (configurations) to one another [21]. There have been various attempts to model biological systems from a computational point of view, including the use of Boolean networks [30], Petri nets [43], the π -calculus [38], interacting state machines [25], L-systems [36] and variants of P systems (membrane systems) [37, 22]. A survey of computational models applied in biology can be found here [21]. These techniques are useful for investigating the qualitative features of the biological systems, as are their stochastic counterparts (e.g., stochastic Petri Nets [26], stochastic P systems [7, 41]) or deterministic systems (so called, MP systems [10]) useful for investigating the quantitative features of computation models.

Having built a model, the goal is typically to *analyse* it, so as to determine the underlying system's properties. Various approaches have been devised for analysing computational models. One widely used method, for example, based on generating the execution traces of a model, is *simulation*. Although the simulation approach is widely applicable, the large number of potential execution paths in models of realistic systems means that we can often exercise only a fraction of the complete trace set using current techniques. Especially for non-deterministic and stochastic systems each state may have more than one possible successor, which means that different runs of the same basic model may produce different outcomes [5]. Consequently, some computational paths may never be exercised, and their conformance to requirements never assessed.

Model checking is another widely recognized approach for analysis and verification of models, which has been successfully applied both to computer systems and biological system models. This technique involves representing each (desired or actual) property as a temporal logic formula, which is then verified against a model. It formally demonstrates the correctness of a system by means of strategically investigating the whole of the model's state space, considering all paths and guaranteeing their correctness [4, 15, 28]. Model checking has advantages over conventional approaches like simulation and testing, because it checks all computational paths and if the specified property is not satisfied it provides useful feedback by generating a counter-example (i.e. execution path) that demonstrates how the failure can occur [28].

Initially, model checking was employed for analysing *transition systems* used for describing discrete systems. A transition system regards time as discrete, and describes a set of states and the possible transitions between them, where each state represents some instantaneous configuration of the system. More recently, model checking has been extended by adding probabilities to state transitions (*probabilistic model checking*); in practice, such systems include discrete-time Markov chains (DTMC), continuous-time Markov chains (CTMC), and Markov decision processes (MDP). Probabilistic models are useful for verifying quantitative features of systems.

Typically, the model checking process comprises the following steps [4, 28]:

1. Describing the system model in a high-level modelling language, so as to provide an unambiguous representation of the input system.

2. Specifying the desired properties (using a property specification language) as a set of logical statements, e.g., temporal logic formulas.
3. Verifying whether each property is valid on the model. For non-probabilistic models the response is either ‘yes’ or ‘no’. For probabilistic systems the response may instead be some estimate of the ‘probability of correctness’.

“Exact” model checking considers whole state spaces while verifying a property, but if the model is relatively large, the verification process can be prohibitively resource intensive and time consuming which is known as ‘state-space explosion’ problem, so this approach can only be applied to a small number of biological models. Nonetheless, the intrinsic power of the approach has gained a good deal of attention from researchers, and model checking has been applied to various biological phenomena, including, for example, gene regulator networks (GRNs) and signal-transduction pathways [7, 13] (see [20] for a recent survey of the use of model checking in systems biology).

To overcome the state-space explosion problem, the *statistical* model checking (SMC) approach does not analyse the entire state space, but instead generates a number of independent simulation traces and uses statistical (e.g., Monte Carlo) methods to generate an approximate measure of system correctness. This approach does not guarantee the absolute correctness of the system, but it allows much larger models be verified (within specified confidence limits) in a faster manner [12, 35, 47, 49]. This approach allows verifying much larger models with significantly improved performance.

The number of tools using statistical model checking has been increasing steadily, as has their application to biological systems [14, 51]. Although the variety of SMC tools gives a certain amount of flexibility and control to users, each model checker has its own specific pros and cons. One tool may support a large set of property operators but perform property verifications slowly, while another may be more efficient at analysing small models, and yet another may excel at handling larger models. In such cases, the user may need to cover all of their options by using more than one model checker, but unfortunately the different SMCs generally use different modelling and property specification languages. Formulating properties using even a single SMC modelling language can be a cumbersome, error-prone, and time wasting experience for non-experts in computational verification (including many biologists), and the difficulties multiply considerably when more than one SMC needs to be used.

In order to facilitate the modelling and analysis tasks, several software suites have been proposed, such as Infobiotics Workbench [8] (based on stochastic P systems [9]) and kPWorkbench framework (based on kernel P systems [17]) [17, 32]. As part of the computational analysis, these tools employ more than one model checker. Currently, they allow only a manual selection of the tools, relying on the user expertise for the selection mechanism. These systems automatically translate the model and queries into the target model checker’s specification language. While this simplifies the checking process considerably, one still has to know which target model checker best suits ones needs, and this requires a significant degree of experience. It is desirable, therefore, to introduce another pro-

cessing layer, so as to reduce human intervention by *automatically* selecting the best model checker for any given combination of P system and property query.

As part of this wider project (Infobiotics Workbench) to provide machine assistance to users, by automatically identifying the best model checker, we evaluate the performance of various statistical model checkers against a pool of biological models. The results reported here can be used to help select the most suitable SMC tools from the available options, by comparing their modelling and property specification languages, capabilities and performances (see also [6]).

Paper structure. We begin in Section 2 by describing some of the most commonly used SMC tools, together with their modelling and property-specification languages. Section 3 compares the usability of these tools in terms of expressibility of their property specification languages. In Section 4 we benchmark the performance of these tools when verifying biological models, and describe the relevant experiment settings. We conclude in Section 5 with a summary of our findings, and highlight open problems that warrant further investigation.

2 A Brief Survey of Current Statistical Model Checkers

In this section, we review some of the most popular and well-maintained statistical model checking tools, together with their modelling and property specification languages.

2.1 Tools

PRISM. PRISM (Probabilistic and Symbolic Model Checker) is a widely-used, powerful probabilistic model checker tool [27, 33]. It has been used for analysing a range of systems including biological systems, communication, multimedia and security protocols and many others [44]. It allows building and analysing several types of probabilistic systems including discrete-time Markov chains (DTMCs) and continuous-time Markov chains (CTMCs) with their ‘reward’ extension. PRISM can carry out both probabilistic model checking based on numerical techniques with exhaustive traversal of model, and statistical model checking with a discrete-event simulation engine [34, 44]. The associated modelling language, the PRISM language (a high-level state-based language), is the probabilistic variant of Reactive Modules [1, 33] (for a full description of PRISM’s modelling language, see [44]), which subsumes several property specification languages, including PCTL, PCTL*, CSL, probabilistic LTL. However, statistical model checking can only be applied to a limited subset of properties; for example, it does not support steady-state and LTL-style path properties.

PRISM can be run via both a Graphical User Interface (GUI) or directly from the command line. Both options facilitate model checking process by allowing to modify a large set of parameters. The command line option is particularly useful when users need to run a large number of models. PRISM is open source software and is available for Windows, Linux and Mac OS X platforms.

PLASMA-Lab. PLASMA-Lab is a software platform for statistical model checking of stochastic systems. It provides a flexible plug-in mechanism which allows users to personalise their own simulator, and it also facilitates distributed simulations [11]. The tool has been applied to a range of problems, such as systems biology, rare events, motion planning and systems of systems [42].

The platform supports four modelling languages: Reactive Module Language (RML) implementation of the PRISM tool language, with two other variants of RML (see Table 1), and Biological Language [11, 42]. In addition, it provides a few simulator plug-ins which enable external simulators to be integrated with PLASMA-Lab, e.g., MATLAB/Simulink. The associated property specification language is based on Bounded Linear Temporal Logic (B-LTL) which bounds the number of states by number of steps or time units.

PLASMA-Lab can be run from a GUI or command line with plug-in system, and while it is not open source it can be embedded within other software programs as a library. It has been developed using the Java programming language, which provides compatibility with different operating systems.

Ymer. Ymer is a statistical model checking tool for verifying continuous-time Markov chains (CTMCs) and generalized semi-Markov processes (GSMPs). The tool supports parallel generation of simulation traces, which makes Ymer a fast SMC tool [48].

Ymer uses the PRISM language grammar for its modelling and property specification language. It employs the CSL formalism for property specification [46].

Ymer can be invoked via a command line interface only. It has been developed using the C/C++ programming language, and the source code is open to the public.

MRMC. MRMC is a tool for numerical and statistical model checking of probabilistic systems. It supports DTMC, CTMC, and using the reward extension of DTMC and CTMC [29].

The tool does not employ a high-level modelling language, but instead requires a sparse matrix representation of probabilities or rates as input. Describing systems in transition matrix format is very hard, especially for large systems, and external tools should be used to automatically generate the required inputs. Both PRISM and Performance Evaluation Process Algebra (PEPA) have extensions which can generate inputs for the MRMC tool [50]. The matrix representation also requires that state labels with atomic propositions be provided in another structure. Properties can be expressed with PCTL and CSL, and with their reward extensions.

MRMC is a command line tool. It has been developed using the C programming language, and the source code is publicly available. Binary distributions for Windows, Linux and Mac OS X are also available [40].

MC2. The MC2 tool enables statistical model checking of simulation traces, and can perform model checking in parallel.

MC2 does not need a modelling language, instead it imports simulation traces generated by external tools for stochastic and deterministic models. The tool uses probabilistic LTL with numerical constraints (PLTLc) for its property specification language, which enables defining numerical constraints on free variables [16].

MC2 can be executed only through its command line interface. The tool was developed using the Java programming interfaces and is distributed as a `.jar` file, therefore the source code is not available to public. The tool is bundled with a Gillespie simulator, called *Gillespie2*. As will be explained in the following section, it is possible to use Gillespie2 to generate simulation traces for the MC2 tool.

2.2 Modelling Languages

As part of the model checking process the system needs to be described in the target SMC modelling language. If the SMC tool relies on external tools, as in the case of MRMC and MC2, users will also have to learn the usage and modelling language of these external tools as well. For example, if users want to use the MRMC tool, they also have to learn how to use PRISM and how to model in the PRISM language.

Table 1 summarises the modelling languages associated with each SMC tool. The PLASMA and Ymer tools provide fair support for the PRISM language. MRMC expects a transition matrix input, but in practice, for large models, it is not possible to generate the transition matrix manually, so an external tool should be used for generating the matrix. MC2 also relies on external tools, because it does not employ a modelling language, instead it expects externally generated simulation traces. If users want to use the MC2 tool, they first have to learn a modelling language and usage of an appropriate simulation tool. For example, in order to use the Gillespie2 simulator as an external tool for MC2, the user should be able to describe their model using the Systems Biology Markup Language (SBML).

Table 1. Modelling languages and external dependency of SMC tools.

SMCs	Modelling Language(s)	Needs an External Tool?	External Tool Modelling Language
PRISM	PRISM language	NO	N/A
PLASMA-Lab	RML of PRISM, Adaptive RML (extension of RML for adaptive systems), RML with importance sampling, Biological Language	NO	N/A
Ymer	PRISM language	NO	N/A
MRMC	Transition matrix	YES, e.g., PRISM	PRISM language
MC2	N/A	YES, e.g., Gillespie2	Systems Biology Markup Language (SBML)

3 Usability

Model checking uses *temporal logics* as property specification languages. In order to query probabilistic features, probabilistic temporal logics should be used. Several probabilistic property specification languages exist, such as Probabilistic Linear Temporal Logic (PLTL) [4], probabilistic LTL with numerical constraints (PLTLc) [16] and Continuous Stochastic Logic (CSL) [2, 3, 34].

In order to ease the property specification process, frequently used properties, called *patterns*, have been identified by previous studies [18, 24]. Patterns represent recurring properties (e.g., something is *always* the case, something is *possibly* the case), and are generally represented by natural language-like keywords. An increasing number of studies have been conducted to identify appropriate pattern systems for biological models [23, 31, 39]. Table 2 lists various popular patterns [24], giving a short description and explaining how they can be represented using existing temporal logic operators.

Table 2. Property patterns

Patterns	Description	Temporal Logic
Existence	ϕ_1 will eventually hold, within the $\bowtie p$ bounds.	$P_{\bowtie p}[F \phi_1]$ or $P_{\bowtie p}[true \text{ U } \phi_1]$
Until	ϕ_1 will hold continuously until ϕ_2 eventually holds, within the $\bowtie p$ bounds.	$P_{\bowtie p}[\phi_1 \text{ U } \phi_2]$
Response	If ϕ_1 holds, then ϕ_2 must hold within the $\bowtie p$ bounds.	$P_{\geq 1}[G(\phi_1 \rightarrow (P_{\bowtie p}[F \phi_2]))]$
Steady-State (Long-run)	In the long-run ϕ_1 must hold, within the $\bowtie p$ bounds.	$S_{\bowtie p}[\phi_1]$ or $P_{\bowtie p}[FG(\phi_1)]$
Universality	ϕ_1 continuously holds, within the $\bowtie p$ bounds.	$P_{\bowtie p}[G \phi_1]$ or $P_{\bowtie(1-p)}[(F(\neg \phi_1))]$

Key. ϕ_1 , and ϕ_2 are state formulas; \bowtie is one of the relations in $\{<, >, \leq, \geq\}$; $p \in [0, 1]$ is a probability with rational bounds; and \bowtie is negation of inequality operators. $P_{\bowtie p}$ is the *qualitative* operator which enables users to query qualitative features, those whose result is either ‘yes’ or ‘no’. In order to query *quantitative* properties, $P_{=?}$ (quantitative operator) can be used to returns a numeric value which is the probability that the specified property is true.

The SMCs investigated here employ different grammar syntaxes for property specification, which makes it harder to use other tools at the same time. Although Ymer uses the same grammar as PRISM, it excludes some operators, such as the **Always** (G) operator. In addition, different SMCs tools may support different sets of probabilistic temporal logics. In the following, we compare the expressibility of their specification languages, by checking if the properties can be defined using just one temporal logic operator, namely directly supported (DS), which will be easier for practitioners to express; or as a combination of multiple operators, indirectly supported (IS); or not supported at all, not supported (NS). Qualitative and quantitative operators, with five property patterns which are identified as widely used by [24], are listed in Table 3.

The PRISM, Ymer and MC2 tools directly support both **Qualitative** and **Quantitative** operators, but MRMC supports only the **Qualitative** operator. While

Table 3. Specifying various key patterns using different SMC tools.

SMCs	Qualitative Operator	Quantitative Operator (P=?)	Existence	Until	Response	Steady-State	Universality
PRISM	DS	DS	DS	DS	NS	NS	DS
PLASMA-Lab	NS	NS	DS	DS	IS	IS	DS
Ymer	DS	DS	DS	DS	NS	NS	IS
MRMC	DS	NS	DS	DS	IS	DS	DS
MC2	DS	DS	DS	DS	IS	IS	DS

Key. DS = Directly Supported; IS = Indirectly Supported; NS = Not Supported.

PLASMA-Lab does not allow these operators to be expressed directly with B-LTL, the verification outputs contain information about the probability of the property, hence users can interpret the results. **Existence**, **Until** and **Universality** properties are directly supported by all SMCs, except that Ymer does not employ an operator for **Universality** patterns (it needs to be interpreted using the **Not** (!) and **Eventually** (*F*) operators, i.e. it is indirectly supported). There is no single operator to represent the **Response** pattern directly, but it is indirectly supported by PLASMA-Lab, MRMC and MC2. The **Steady-State** pattern can be either represented by one operator, *S*, or two operators, *F* and *G*. Only the MRMC tool employs the *S* operator to allow **Steady-State** to be expressed directly, while PLASMA-Lab and MC2 allow it to be expressed indirectly.

4 Experimental Findings

The wide variety of SMC tools gives a certain flexibility and control to users, but practitioners need to know which of the tools is the most suitable for their particular models and queries. The expressive power of the associated modelling and specification languages is not the only criterion, because SMC performance may also depend on the nature of the models and property specifications. We have therefore conducted a series of experiments to determine the capabilities and performances of the most commonly used tools [6]. The experiments are conducted on an Intel i7-2600 CPU @ 3.40GHz 8 cores, with 16GB RAM running on Ubuntu 14.04.

We tested each of the five tools against a representative selection of 465 biological models (in SBML format) taken from the BioModels database [19] (as modified in [45] to fix the stochastic rate constants of all reactions to 1). The models tested ranged in size from 2 species and 1 reaction, to 2631 species and 2824 reactions. Figure 1 shows the distribution of models size, we take “size” to be the product of species count and reaction count. X-axis (log scale) indicates the model size and Y-axis represents the frequency of models with their sizes represented on the X-axis.

Each tool/model pair was tested against five different property specification patterns [24], namely **Existence**, **Until**, **Response**, **Steady-State** and **Universality**. We have developed a tool for translating SBML models to SMC modelling lan-

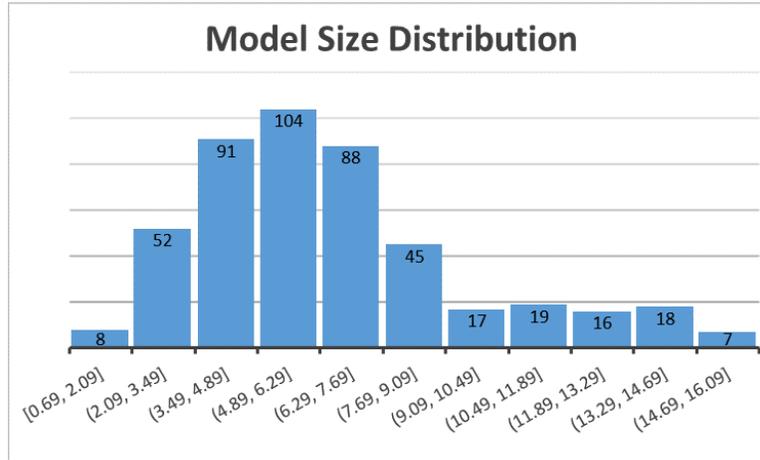


Fig. 1. The distribution of models size in the logarithmic scale.

guages, and translating property patterns to the corresponding SMC specification languages. For each SMC, the number of simulation traces was set to 500, and the depth of each trace was set to 5000.

The time required for each run is taken to be the combined time required for model parsing, simulation and verification. Each SMC/model/pattern combination was tested three times, and the figures reported here give the average total time required. When an SMC depends on external tools, we also added the external tool execution time into the total execution time. In particular, therefore, the total times reported for verifying models with MRMC and MC2 tools are not their execution times only, but include the time consumed for generating transition matrices and simulation traces, respectively. We used the PRISM tool for generating transition matrices requested by MRMC, and the Gillespie2 for generating simulation traces utilised by MC2. When the external tool failed to generate the necessary input for its corresponding SMC, we have recorded the SMC as being incapable of verifying the model. In order to keep the experiment tractable, when an SMC required more than 1 hour to complete the run, we halted the process and again recorded the model as unverifiable.

Table 4 shows the experiment results. The SMCs and the property patterns are represented in the first column and row, respectively. The *Verified* columns under each pattern show the number of models that could be verified by the corresponding SMC. The *Fastest* column shows the number of models for which the corresponding SMC was the fastest tool.

The results show that SMC tool capabilities vary depending on the queried properties. For example, PRISM was only able to verify 337 models against *Existence*, and 435 and 370 models against *Until* and *Universality*, respectively. The main reason PRISM failed to verify *all* of the models is that it expects user to increase the depth of the simulation traces, otherwise it cannot verify the

Table 4. The number of model/pattern combinations verified by each SMC tool.

	Existence		Until		Response		Steady- -State		Universality	
	Verified	Fastest	Verified	Fastest	Verified	Fastest	Verified	Fastest	Verified	Fastest
PRISM	337	15	435	84	NS	NS	NS	NS	370	57
PLASMA-Lab	465	143	465	54	465	390	465	392	465	80
Ymer	439	304	439	324	NS	NS	NS	NS	439	325
MRMC	75	0	72	0	75	17	57	11	77	0
MC2	458	3	458	3	458	58	458	62	458	3

Key. NS = Not Supported.

unbounded properties with a reliable approximation. In contrast, PLASMA-Lab was able to verify all of the models within 1 hour. Ymer could verify 439 models for those patterns it supports, thus failing to complete 26 models in the time available. MRMC was able to verify relatively few models, because it relied on the PRISM model checker to construct the model and export the associated transition matrices. Especially for relatively large models PRISM crashed while generating these matrices (we believe this is related to its CU Decision Diagram (CUDD) library). MC2 was able to verify 458 models against all of the patterns tested, and only failed for 7 of them.

The second column of the patterns shows the number of models which were verified by the corresponding model checker tools. The distribution of models size across the fastest model checkers for different patterns are shown in the following set of violin plots (Figures 2 – 6). Each of the inner swarm points represents a model. X-axis represents the logarithmic scale of models size. For the models in the white background region, we can uniquely identify the fastest SMC tool for their verification, whereas for the models in grey background region the fastest model checker is not clear.

Ymer was the fastest for most model/pattern pairs (where those patterns were supported). However, it is the fastest tool only for verification of relatively small size models. Ymer was the fastest for verifying 304 models against Existence pattern, the minimum model size verified by Ymer was 2, maximum 2128, mean 256.8 and median 137.5. It was the fastest tool for larger number of models, 324 (min = 2, max = 2128, mean = 312.9, median = 144), against Until pattern verification, and 325 models (min = 2, max = 2346, mean = 335, median = 144) against Universality pattern verification. PLASMA-Lab is the fastest tool for relatively large size models. It was the fastest tool for verifying 143 models (min = 380, max = 7429944, mean = 464498.9, median = 11875) against Existence pattern, 54 models (min = 1224, max = 7429944, mean = 837193.5, median = 288162) against Until pattern, and 80 models (min = 575, max = 7429944, mean = 773247.5, median = 43143) against Universality pattern verification. It did particularly well against Response, 390 models (min = 12, max = 7429944, mean = 170734.5, median = 604.5) and Steady-State patterns, 392 models (min = 9, max = 7429944, mean = 169862.1, median = 600), where it was only competing

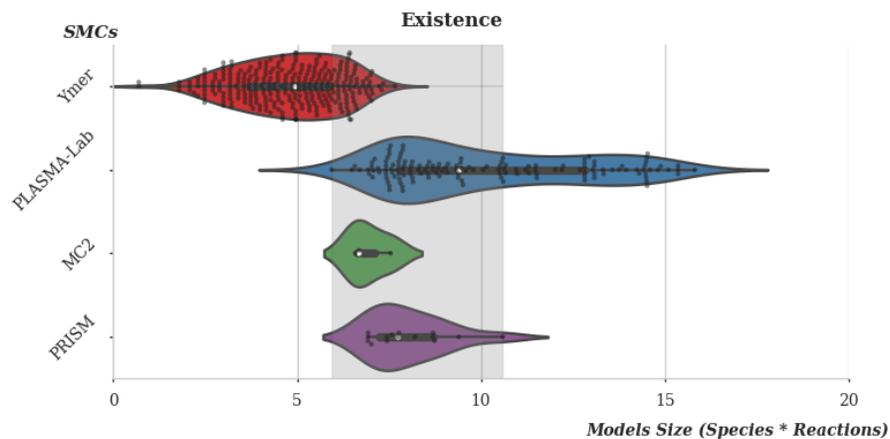


Fig. 2. The distribution of models size across fastest SMC tools for Existence pattern verification.

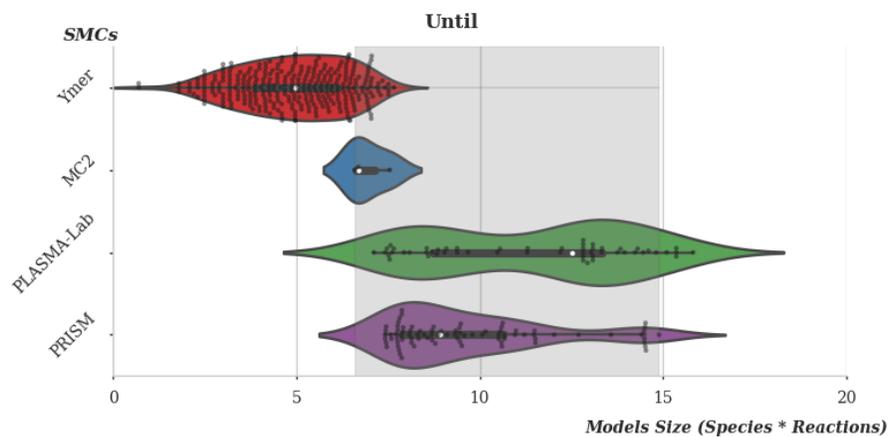


Fig. 3. The distribution of models size across fastest SMC tools for Until pattern verification.

with MRMC and MC2. PRISM is generally the fastest tool for medium to large size models. It was the fastest only for 15 models (min = 1023, max = 39770, mean = 5860.9, median = 2304) against Existence pattern verification, but it was able to verify larger number of models, 84 (min = 1665, max = 2928904, mean = 253327.3, median = 7395), against Until pattern verification and 57 models (min = 960, max = 1633632, mean = 92998.4, median = 3364) against Universality pattern verification. MC2 (with Gillespie2) is the fastest for relatively small size models. It could verify only 3 models (min = 722, max = 1892, mean = 1138,

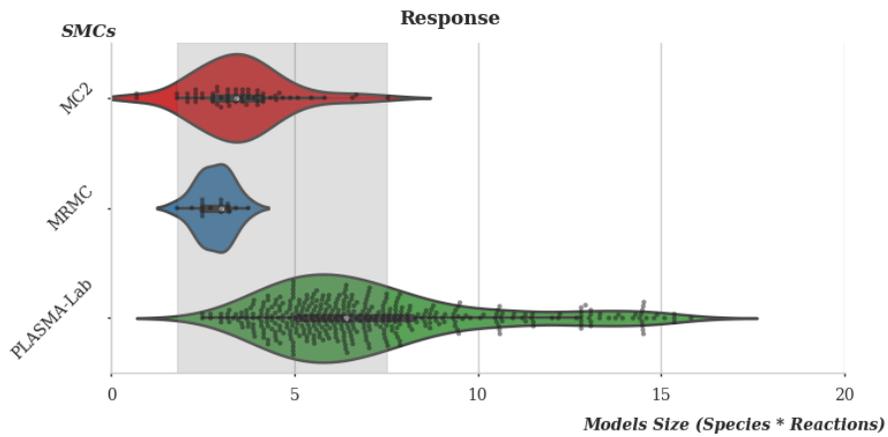


Fig. 4. The distribution of models size across fastest SMC tools for Response pattern verification.

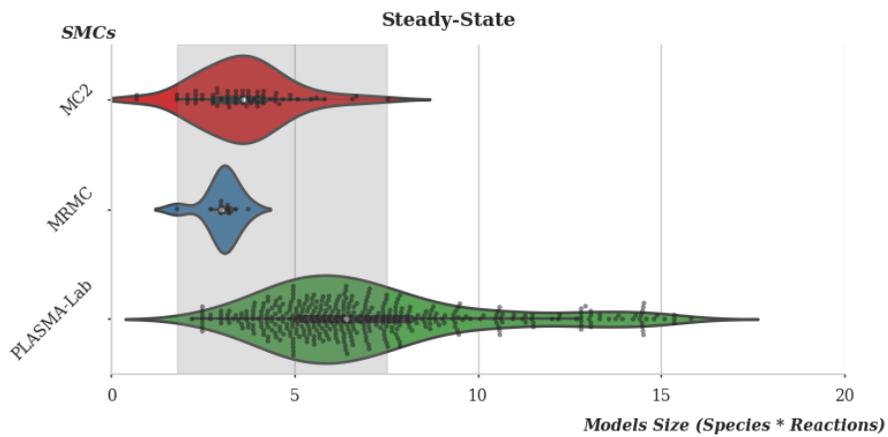


Fig. 5. The distribution of models size across fastest SMC tools for Steady-State pattern verification.

median = 800) against Existence, Until and Universality patterns, although it did better with 58 models (min = 2, max = 1892, mean = 103.2, median = 30) against Response pattern, and 62 models (min = 2, max = 1892, mean = 105.9, median = 36) against Steady-State patterns. Finally, MRMC (with PRISM dependency) was slower than other tools for Existence, Until and Universality patterns verification, but did better handling Response (fastest for 17 models: min = 6, max = 42, mean = 18.5, median = 20) and Steady-State (fastest for 11 models: min = 6, max = 42, mean = 22.3, median = 20).

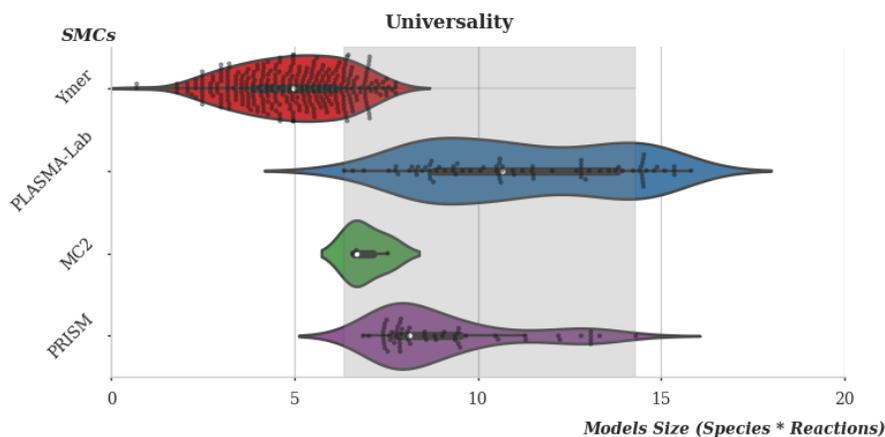


Fig. 6. The distribution of models size across fastest SMC tools for *Universality* pattern verification.

As we stated previously, the background color of Figures 2 – 6 gives an indication of whether the fastest model checker can be identified for the models within a region of the graph, that is, for models in the white background region, the fastest SMC tool can be identified, but the models in grey background region it is less clear-cut. For verification of *Existence* pattern, we can uniquely identify the fastest SMC tool for both the 232 smallest models (size ranging from 2 to 380), and the 55 largest models (size = 39984 to 7429944), namely Ymer and PLASMA-Lab respectively, but for remaining 178 medium-sized models (size = 380 to 39770), there is no obvious ‘winner’. Similarly, for *Until* pattern verification, the smallest 283 models (size ranging from 2 to 714), and only for the 5 largest models (size = 3605380 to 7429944) we can identify the fastest SMC tool (Ymer and PLASMA-Lab respectively), but there are more than one candidates for remaining 177 medium-sized models (size = 722 to 2928904). Despite, we have only three SMC tools, namely PLASMA-Lab, MRMC and MC2, which support the verification of *Response* and *Steady-State* patterns, their performance on small and medium size models are close to each other, which makes harder to identify the fastest tool. Therefore, only for the smallest 4 models (size = 2 to 6) and for the largest 128 models (size= 1927 to 7429944) the fastest tool (MC2 and PLASMA-Lab respectively) can be identified. Lastly, for *Universality* pattern verification, the fastest SMC tool for both smallest 262 models (size=2 to 572) and largest 17 models (size =1823582 to 7429944), Ymer and PLASMA-Lab respectively, can be identified, for the remained 186 medium size models we cannot assign a unique model checker tool.

5 Conclusion

The experimental results clearly show that certain SMC tools are best for certain tasks, but there are also situations where the best choice of SMC is far less clear-cut, and it is not surprising that users may struggle to select and use the most suitable SMC tool for their needs. Users need to consider the modelling language of tools and the external tools they may rely on, and need detailed knowledge as to which property specification operators are supported, and how to specify them. Even then, the tool may still fail to complete the verification within a reasonable time, whereas another tool might be able to run it successfully.

These factors make it extremely difficult for users to know which model checker to choose, and point to a clear need for automation of the SMC-selection process. We are currently working to identify novel methods and algorithms to automate the selection of best SMC tool for a given computational model (more specifically for P system models) and property patterns. We aim to enable the integration of our methods within larger software platforms, e.g., IBW and kP-Workbench, and while this is undoubtedly a challenging task, we are encouraged by recent developments in related areas, e.g., the automatic selection of stochastic simulation algorithms [45].

References

1. Alur, R., Henzinger, T.A.: Reactive modules. *Form. Methods Syst. Des.* 15, 7–48 (Jul 1999), <http://dx.doi.org/10.1023/A:1008739929481>
2. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic* 1(1), 162–170 (Jul 2000), <http://doi.acm.org/10.1145/343369.343402>
3. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering* 29(6), 524–541 (June 2003)
4. Baier, C., Katoen, J.P.: *Principles of model checking*. The MIT Press (2008)
5. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipate, F.: High performance simulations of kernel P systems. 2014 IEEE 16th International Conference on High Performance Computing and Communications (HPCC) (2014)
6. Bakir, M.E., Stannett, M.: Selection criteria for statistical model checking. In: Gheorghe, M., Konur, S. (eds.) *Proceedings of the Workshop on Membrane Computing WMC 2016*, Manchester (UK), 11-15 July 2016. pp. 55–57 (2016), <http://bradscholars.brad.ac.uk/handle/10454/8840>, Available as: Technical Report UB-20160819-1, University of Bradford
7. Bernardini, F., Gheorghe, M., Romero-Campero, F.J., Walkinshaw, N.: A hybrid approach to modeling biological systems. In: Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *Membrane Computing, Lecture Notes in Computer Science*, vol. 4860, pp. 138–159. Springer Berlin Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-77312-2_9
8. Blakes, J., Twycross, J., Romero-Campero, F.J., Krasnogor, N.: The Infobiotics Workbench: An integrated in silico modelling platform for systems and synthetic biology. *Bioinformatics* 27(23), 3323–3324 (Dec 2011)

9. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F.J., Krasnogor, N., Gheorghe, M.: Infobiotics Workbench: A P systems based tool for systems and synthetic biology. In: Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.) *Applications of Membrane Computing in Systems and Synthetic Biology, Emergence, Complexity and Computation*, vol. 7, pp. 1–41. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-03191-0_1
10. Bollig-Fischer, A., Marchetti, L., Mitrea, C., Wu, J., Kruger, A., Manca, V., Drăghici, S.: Modeling time-dependent transcription effects of her2 oncogene and discovery of a role for e2f2 in breast cancer cell-matrix adhesion. *Bioinformatics* 30(21), 3036–3043 (2014)
11. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma-lab: A flexible, distributable statistical model checking library. In: Joshi, K., et al. (eds.) *Proceedings of Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013, LNCS*, vol. 8054, pp. 160–164. Springer-Verlag (2013)
12. Buchholz, P.: A new approach combining simulation and randomization for the analysis of large continuous time Markov chains. *ACM Trans. Model. Comput. Simul.* 8(2), 194–222 (Apr 1998), <http://doi.acm.org/10.1145/280265.280274>
13. Carrillo, M., Góngora, P.A., Rosenblueth, D.A.: An overview of existing modeling tools making use of model checking in the analysis of biochemical networks. *Frontiers in Plant Science* 3(155), 1–13 (2012)
14. Cavaliere, M., Mazza, T., Sedwards, S.: Statistical model checking of membrane systems with peripheral proteins: Quantifying the role of estrogen in cellular mitosis and DNA damage. In: *Applications of Membrane Computing in Systems and Synthetic Biology, Emergence, Complexity and Computation*, vol. 7, pp. 43–63. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-03191-0_2
15. Clarke, E.M., Grumberg, O., Peled, D.: *Model checking*. MIT Press (1999)
16. Donaldson, R.;Gilbert, D.: A Monte Carlo model checker for Probabilistic LTL with numerical constraints. Tech. rep., University of Glasgow, Department of Computing Science (2008)
17. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierla, L.: Model checking kernel P systems. In: *Membrane Computing, Lecture Notes in Computer Science*, vol. 8340, pp. 151–172. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-642-54239-8_12
18. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: *Patterns in Property Specifications for Finite-state Verification*, pp. 411–420. ICSE '99, ACM, New York, NY, USA (1999)
19. The European Bioinformatics Institute. <http://www.ebi.ac.uk/>, [Online; accessed 25/09/16]
20. Fisher, J., Piterman, N.: *Model checking in biology*, pp. 255–279. Springer Verlag (2014)
21. Fisher, J., Henzinger, T.A.: Executable cell biology. *Nat Biotech* 25(11), 1239–1249 (2007)
22. Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.): *Applications of Membrane Computing in Systems and Synthetic Biology, Emergence, Complexity and Computation*, vol. 7. Springer International Publishing (2014)
23. Gheorghe, M., Konur, S., Ipate, F., Mierla, L., Bakir, M.E., Stannett, M.: An integrated model checking toolset for kernel P systems. In: Rozenberg, G., Salomaa, A., Sempere, M.J., Zandron, C. (eds.) *Membrane Computing: 16th International*

- Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers, pp. 153–170. Springer International Publishing, Cham (2015)
24. Grunske, L.: Specification Patterns for Probabilistic Quality Properties, pp. 31–40. ICSE '08, ACM, NY, USA (2008)
 25. Harel, D.: Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8(3), 231–274 (1987), [http://dx.doi.org/10.1016/0167-6423\(87\)90035-9](http://dx.doi.org/10.1016/0167-6423(87)90035-9)
 26. Heiner, M., Gilbert, D., Donaldson, R.: Petri nets for systems and synthetic biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) *Proceedings of the Formal Methods for the Design of Computer, Communication, and Software Systems 8th International Conference on Formal Methods for Computational Systems Biology (SFM 2008)*. LNCS, vol. 5016, pp. 215–264. Springer-Verlag, Berlin, Heidelberg (2008)
 27. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 441–444. Springer Berlin Heidelberg (2006)
 28. Huth, M., Ryan, M.: *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edn. (2004)
 29. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. In: *Quantitative Evaluation of Systems (QEST)*. pp. 167–176. IEEE Computer Society (2009)
 30. Kauffman, S.A.: Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology* 22, 437–467 (1969)
 31. Konur, S., Gheorghe, M.: A property-driven methodology for formal analysis of synthetic biology systems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 12(2), 360–371 (March 2015)
 32. kPWorkbench. <http://kpworkbench.org/>, [Online; accessed 25/09/16]
 33. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: *Computer performance evaluation: modelling techniques and tools*, pp. 200–204. Springer Berlin Heidelberg (2002)
 34. Kwiatkowska, M., Norman, G., Parker, D.: *Stochastic Model Checking*, pp. 220–270. SFM'07, Springer-Verlag, Berlin, Heidelberg (2007)
 35. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: *Proceedings of Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4*. pp. 122–135. Springer, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-16612-9_11
 36. Lindenmayer, A., Jürgensen, H.: Grammars of development: Discrete-state models for growth, differentiation, and gene expression in modular organisms. In: *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*, pp. 3–21. Springer, Berlin, Heidelberg (1992), http://dx.doi.org/10.1007/978-3-642-58117-5_1
 37. Manca, V.: *Infobiotics: Information in Biotic Systems, Emergence, Complexity and Computation*, vol. 3. Springer International Publishing (2013)
 38. Milner, R.: *Communicating and mobile systems: The Pi-calculus*. Cambridge University Press, New York, NY, USA (1999)
 39. Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., de Jong, H.: Temporal logic patterns for querying dynamic models of cellular interaction networks. *Bioinformatics* 24(16), i227–i233 (Aug 2008), <http://dx.doi.org/10.1093/bioinformatics/btn275>
 40. Markov Reward Model Checker (MRMC). <http://www.mrmc-tool.org/>, [Online; accessed 18/02/15]

41. Pérez-Jiménez, M.J., Romero-Campero, F.J.: P systems, a new computational modelling tool for systems biology. In: Priami, C., Plotkin, G. (eds.) Transactions on Computational Systems Biology VI, pp. 176–197. Springer, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11880646_8
42. Plasma-Lab. <https://project.inria.fr/plasma-lab/>, [Online; accessed 18/02/15]
43. Reisig, W.: The basic concepts. In: Understanding Petri Nets, pp. 13–24. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-33278-4_2
44. Probabilistic and Symbolic Model Checker (PRISM). <http://www.prismmodelchecker.org/>, [Online; accessed 08/01/15]
45. Sanassy, D., Widera, P., Krasnogor, N.: Meta-stochastic simulation of biochemical models for systems and synthetic biology. ACS Synthetic Biology 4(1), 39–47 (2015), <http://dx.doi.org/10.1021/sb5001406>, PMID: 25152014
46. Ymer website. <http://www.tempastic.org/ymer/>, [Online; accessed 25/8/15]
47. Younes, H., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. International Journal on Software Tools for Technology Transfer (STTT) 8(3), 216–228 (2006)
48. Younes, H.L.S.: Ymer: A Statistical Model Checker, pp. 429–433. CAV’05, Springer-Verlag, Berlin, Heidelberg (2005)
49. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Proceedings of Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, pp. 223–235. Springer, Berlin, Heidelberg (2002), http://dx.doi.org/10.1007/3-540-45657-0_17
50. Zapreev, I.S., Jansen, C.: Markov reward model checker manual, http://www.mrmc-tool.org/downloads/MRMC/Specs/MRMC_Manual.pdf
51. Zuliani, P.: Statistical model checking for biological applications. International Journal on Software Tools for Technology Transfer 17(4), 527–536 (2014), <http://dx.doi.org/10.1007/s10009-014-0343-0>

Chapter 7

Automating Statistical Model Checking Tool Selection

The previous chapter showed that the performance of statistical model checking tools varies per model size and pattern type, and concluded by emphasising the need to automate the SMC tools selection process.

Fourth study: Mehmet Emin Bakir, Savas Konur, Marian Gheorghe, Natalio Krasnogor, and Mike Stannett. Performance benchmarking and automatic selection of verification tools for efficient analysis of biological models. Unpublished, 2017.

In this paper, we extended the previous study [11] by benchmarking larger sets of biological models and property patterns, i.e. 675 models and 11 patterns. The instantiation of the property patterns are provided in Appendix B. Please note that for our study the actual instantiation is not very important, as we provide the same property instance and models to all SMC tools and we evaluate their performance. To predict the best SMC tool, initially, we used topological properties of species and reaction dependency graphs, e.g. the number of vertices and edges, as initially proposed by [104] for stochastic simulation algorithm performance prediction. Additionally, we introduced a number of custom (non-graph related) features, such as the number of variables to be updated when a reaction triggers. The final set contains both graph and non-graph related features.

We demonstrated that they are computationally efficient and yet can notably increase the prediction accuracy. Then, we provided the final feature set as input to a number of machine learning algorithms. The winner algorithm of each pattern could successfully predict the fastest SMC tool with over 90% accuracy. We developed a software utility tool, SMC Predictor, which automatically matches the best statistical model checkers to a given and property pattern. Finally, we demonstrated that by using our methodology users can save a significant amount of time. The study in this chapter has been written in a scientific manuscript format which is a preliminary study for a journal paper. It presents an initial, and in some respects more comprehensive, account of work subsequently accepted (after formal submission of this thesis) for publication in revised form as [13].

Performance Benchmarking and Automatic Selection of Verification Tools for Efficient Analysis of Biological Models

Mehmet Emin Bakir,^{*,†} Savas Konur,^{*,‡} Marian Gheorghe,^{*,‡} Natalio Krasnogor,^{*,¶}
and Mike Stannett^{*,†}

[†]*Department of Computer Science, University of Sheffield, Sheffield S1 4DP, UK*

[‡]*School of Electrical Engineering & Computer Science, University of Bradford,
Bradford BD7 1DP, UK*

[¶]*Interdisciplinary Computing and Complex BioSystems (ICOS) Research Group, School of
Computing Science, Newcastle University, Newcastle NE1 7RU, UK*

E-mail: mebakir1@sheffield.ac.uk; s.konur@bradford.ac.uk; m.gheorghe@bradford.ac.uk;
natalio.krasnogor@newcastle.ac.uk; m.stannett@sheffield.ac.uk

Abstract

Simulation and verification, particularly model checking, are two computational techniques, which have been widely used in the analysis of biological systems. Although model checking enables the discovery of systems properties, it suffers from performance issues since it employs mathematical techniques that are computationally very demanding. *Statistical Model Checking* (SMC) has been introduced to alleviate this issue by replacing mathematical and numerical analysis with the simulation approach, which is computationally less demanding. Namely, SMC combines simulation and model checking by leveraging the speed of simulation with the comprehensive analytic capacity of

model checking. The success of SMC has prompted researchers to implement a number of SMC tools and apply them to various fields, including systems biology, membrane computing and synthetic biology. However, while the availability of multiple tools gives flexibility and fine-tuned control over model analysis, these tools generally have different modelling and specification languages and support different property operators. More importantly, they differ in their performance characteristics. However, choosing the most efficient tool for any given model requires a significant degree of experience, and in most of the cases, it is challenging to predict the right one.

The objective of this study is to automate SMC tool selection to streamline biological model analysis. Our results suggest it is possible to successfully predict the best SMC tool for a given biological model with over 90% accuracy.

Abbreviations

Statistical Model Checking (SMC), Stochastic Simulation Algorithms (SSA)

Keywords

statistical model checking, synthetic biology, *in silico*, biological models, verification

1 Introduction

In order to handle the complex structure and the dynamic functionality of biological systems, researchers have developed a number of machine-executable mathematical and computational models to help them understand fine-grained spatial and temporal biological behaviours (1). The executable nature of these models enables researchers to conduct *in silico* experiments, which are generally faster, cheaper and more reproducible than the analogous wet-lab experiments, and only validated computational models need be implemented as wet-lab experiments. However, the relevance of any particular *in silico* experiment depends on whether the

methods used to validate the models can scale efficiently to handle large problem instances while maintaining the precision of the results obtained.

Simulation and model checking (2), which is an algorithmic formal verification technique, are two powerful techniques used for analysing computational models. Each has its own advantages and disadvantages. *Simulation* works by executing the model repeatedly, and analysing the result. Each run of the system can be performed relatively quickly, but – especially in large, non-deterministic models – it is generally not possible to guarantee that we will execute every possible computation path. In contrast, *model checking* works by representing desirable properties of the model using formal mathematical logic, and then verifying whether or not the model satisfies the corresponding formal specification. This involves checking the model’s entire state space *exhaustively*. As a result, unlike simulation, model checking allows discovering novel knowledge about system properties. However, the very well-known *state-space explosion problem* associated with large non-deterministic systems (as a result of exhaustive analysis using mathematical and numerical methods) means that the approach can be applied effectively to only a small number of biological models.

Statistical Model Checking (SMC) (3) has been introduced to alleviate the state-explosion problem issue by replacing mathematical and numerical analysis with the simulation approach, which is computationally less demanding. Namely, SMC combines simulation and model checking by leveraging the speed of simulation with the comprehensive analytic capacity of model checking.

Like standard model checking, SMC is essentially a three-step process, involving: (i) mapping a system to the model checker’s modelling language; (ii) specifying desired properties using an appropriate property specification language; and (iii) verifying whether the properties are valid for the model (4, 5). However, since it is not generally possible to execute every possible run of the system in a reasonable amount of time, instead it analyses a random sample of execution paths and then uses statistical methods – for example, Monte Carlo methods – to determine the “approximate” correctness of the model to within

identifiable confidence limits. The greatly reduced number of executions enables verification of larger models at far lower computational cost, albeit by introducing a small amount of uncertainty.

The success of statistical model checking has prompted researchers to implement a number of SMC tools and apply them to various fields, including computer systems, systems biology(6), membrane computing (7) and synthetic biology (8, 9). However, while the availability of multiple variants of these tools and algorithms can allow considerable flexibility and fine-tuned control over the analysis of specific models, it can also be difficult for synthetic biologists to acquire the knowledge needed to identify clearly which tools are most appropriate in any given situation. This is because

- different tools typically use different modelling and property specification languages, and support the analysis of different collections of properties – users therefore need to familiarise themselves with a range of different technologies;
- while some tools are self-contained, others depend on the use of external third party applications for pre-processing, which means that the users need to learn the techniques involved in using these other tools as well;
- the performance characteristics of any given tool may vary significantly according to the verifications being performed. Where one tool successfully verifies a model’s properties efficiently, another may fail; and any given tool may succeed in validating certain properties of a model, but fail to verify others.

Consequently, researchers generally need to use more than one tool to cover their needs, but may easily lack the knowledge required to do so. Verification using even a single SMC tool can be an error-prone and time-consuming process, and the difficulties can multiply considerably when more than one tool is involved. Clearly, this is a very tedious task for non-experts, including biologists.

In response to this problem, a number of software suites have recently been developed to facilitate the model checking process, such as SMBIONET (10), BIOCHAM system (11), BIO-PEPA ECLIPSE WORKBENCH (12), GENETIC NETWORK ANALYZER (GNA) (13), KPWORKBENCH (14–16) and INFOBIOTICS WORKBENCH (17, 18). These tools integrate a number of model checking tools in a single platform. They abstract the complexity of the model checker requirements by providing a high-level modelling and property specification language and by enabling users to select a target model checker for verification. These systems internally translate from high-level languages to those required by the target model checker. While this noticeably simplifies the model checking process, users still have to know which of the target model checkers is best for the model and properties under consideration, which again requires a significant degree of verification experience. Consequently, it is currently common practice to verify models with a randomly picked SMC tool, whereas another tool might actually enable verification to progress significantly faster.

These complications can discourage non-experts – potentially the majority of professional biologists – from building and analysing large system models. It is therefore desirable to have a system which can automatically identify the fastest SMC for a given model and property. Automating this SMC selection process will not only significantly reduce the total time and effort employed in model verification, but will also enable more precise verifications of complex models while keeping the verification time tractable. In consequence, we believe, a deeper understanding of biological system dynamics is more likely to be achieved.

Generally speaking, most of the time involved in executing the SMC verification process consists in generating simulation traces and verifying the queried property. Therefore, those factors which most affect simulation and verification time are likely to be the most useful features to consider when predicting the fastest SMC for a given situation. In a recent work (5), we have shown that the type of property (the *property pattern*) being queried is an effective discriminant between SMCs, and is also a significant factor determining the verification time. Regarding the simulation time, Sanassy et al. (19) have shown that the *network char-*

acteristics of models are determining factors for the simulation time, and that by analysing network properties of species and reaction dependency graphs, the fastest Stochastic Simulation Algorithms (SSAs) for a given model can be predicted prior to the model’s execution.

Contributions. The objective of this work programme is to benchmark the performances SMC tools and to automatically match the fastest SMC tool to any given model and property pattern by using machine learning. The main contributions reported here are:

- We propose a set of model features which can be used for SMC prediction, reduce the computation time required for feature extraction, and increase the prediction accuracy.
- We extend our previous performance benchmark study (5) by verifying 675 biological models against 11 well-known property patterns with five SMC tools.
- We have explored the use of 7 machine learning algorithms to predict the fastest SMC tool by analysing the queried property and the model features, and report our findings.
- To the best of our knowledge, this is the first published attempt to predict the fastest SMC tools by grouping the tools’ performance with different property patterns.
- We implement our approach and develop a software tool SMC Predictor that predicts the fastest SMC tool based on a given model and property query.

We have demonstrated that our approach can predict the fastest SMC tool, with over 90% of accuracy for each property pattern, and can therefore help users to save a significant amount of time for model verification.

Paper structure. In Section “Statistical Model Checking (SMC) Tools ” we describe some of the SMC tools we investigated in this study and the commonly used property patterns. Section “Experiments” explains our experimental findings: in Section“ Feature Selection” we discuss several new model features, and compare their usage against the network features for identifying the fastest SSA predictors; in Section “Performance Benchmarking of SMC

Tools” the performance comparison of SMC tools for different property types is presented; in Section “Automating SMC Tool Prediction” the prediction accuracies of various machine learning algorithms are compared; in Section “Performance Gain and Loss” we demonstrate the amount of time that a user can save by using our automated prediction system; and in “SMC Predictor Tool” we present the architecture of our software tool, SMC Predictor. Section “Conclusion” concludes with a summary of our findings, and discusses possible future directions.

2 Statistical Model Checking (SMC) Tools

In this section we briefly introduce the five widely used SMC tools considered in our experimental analysis: PRISM, PLASMA-Lab, Ymer, MRMC and MC2. They have been used for analysing a wide range of systems, including computer, network and biological systems. For comprehensive analysis of the usability and capacity of these tools and their application for biological systems we refer the reader to (5, 20–23).

PRISM (Probabilistic and Symbolic Model Checker) is a popular and well maintained probabilistic model checker tool (24, 25). PRISM implements both probabilistic model checking based on numerical techniques with exhaustive analysis of model (called numeric model checking) and statistical model checking using an internal discrete-event simulation engine (26, 27). PLASMA-Lab is another statistical model checker for analysing stochastic systems (21). In addition to its internal simulator, it also provides a plugin mechanism to users, allowing them to integrate custom simulators into the PLASMA-Lab platform. Ymer is one of the first tools that implemented statistical model checking algorithms – its ability to parallelise the execution of simulation runs makes it a relatively fast tool (28). MRMC (Markov Reward Model Checker) is another tool which can support both numeric and statistical model checking of probabilistic systems. It is worth noting, however, that MRMC does not provide a high-level modelling language, but instead expects a low level

transition matrix to be provided as input. In practice, this means that MRMC users generally need to use an external tool, such as PRISM, to generate the required inputs (29). Finally, MC2 (Monte Carlo Model Checker) enables statistical model checking over simulation paths. While this tool does not have an internal simulator, and instead uses simulation paths of external simulators, it can parallelise the path evaluation process (22).

For experimental purposes, we used version 2.0 beta2 of MC2 with the Gillespie2 simulator (which is bundled with the tool), to generate the required simulation traces. We also used: PRISM, version 4.2.1; PLASMA-Lab, version 1.3.2, with default settings and simulator; and MRMC version 1.5 together with PRISM 4.2.1 for MRMC performance benchmarking.

2.1 Property Patterns

Model checking uses variants of *temporal logics*, for example Continuous Stochastic Logic (CSL) (27, 30, 31) and Probabilistic Linear Temporal Logic (PLTL) (32), to specify desired system properties and requirements. However, this is a tedious task, because expressing the requirements in such formal specifications requires expert knowledge of formal languages. In order to facilitate the property specification process, various frequently used property types (*patterns*) have been identified in previous studies (33, 34). Recently, more studies have been conducted to identify pattern systems that are particularly appropriate for biological models (35–38).

We identified 11 popular property patterns in the literature for use in our experiment settings. Table 1 lists and describes these patterns, shows how to represent them using temporal logic operators, and shows which SMC tools currently support the relevant pattern expressions. Various standard patterns (Existence, Always, Precedes, Never, Until, Release and Weak Until) are supported by all five SMC tools, whereas the Next pattern is supported by all tools except Ymer. The Steady State pattern is supported only by PLASMA-Lab, MRMC and MC2. The Infinitely Often pattern is only supported by PLASMA-Lab and MC2.

Table 1: Property patterns.

Patterns	Description	Temporal Logic	Supported by
Eventually (Existence)	Within the $\bowtie p$ bounds, ϕ_1 will eventually hold.	$P_{\bowtie p}[F \phi_1]$ or $P_{\bowtie p}[true \text{ U } \phi_1]$	PRISM, PLASMA-Lab, Ymer, MRMC and MC2
Always (Universality)	Within the $\bowtie p$ bounds, ϕ_1 continuously holds.	$P_{\bowtie p}[G \phi_1]$ or $P_{\bowtie(1-p)}[(F (\neg \phi_1))]$	PRISM, PLASMA-Lab, Ymer, MRMC and MC2
Follows (Response)	Within the $\bowtie p$ bounds, If ϕ_1 holds, then ϕ_2 must hold.	$P_{\geq 1}[G (\phi_1 \rightarrow (P_{\bowtie p} [F \phi_2]))]$	PLASMA-Lab, MRMC and MC2
Precedes	Within the $\bowtie p$ bounds, ϕ_1 precedes or activates ϕ_2 .	$P_{\bowtie p}[\neg \phi_2 \text{ W } \phi_1]$ or $P_{\bowtie(1-p)}[\neg \phi_1 \text{ U } (\neg \phi_1 \ \& \ \phi_2)]$	PRISM, PLASMA-Lab, Ymer, MRMC and MC2
Never (Absence)	Within the $\bowtie p$ bounds, ϕ_1 will never hold.	$P_{\bowtie(1-p)}[(F(\phi_1))]$	PRISM, PLASMA-Lab, Ymer, MRMC and MC2
Steady-State (Long-run)	Within the $\bowtie p$ bounds, in the long-run ϕ_1 must hold	$S_{\bowtie p}[\phi_1]$ or $P_{\bowtie p}[FG (\phi_1)]$	PLASMA-Lab, MRMC and MC2
Until	Within the $\bowtie p$ bounds, ϕ_1 holds continuously until ϕ_2 eventually hold.	$P_{\bowtie p}[\phi_1 \text{ U } \phi_2]$	PRISM, PLASMA-Lab, Ymer, MRMC and MC2
Infinitely Often (Recurrence)	Within the $\bowtie p$ bounds, ϕ_1 repeatedly holds.	$P_{\bowtie p}[G (F \phi_1)]$	PLASMA-Lab MC2
Next	Within the $\bowtie p$ bounds, ϕ_1 will hold in the next state.	$P_{\bowtie p}[X \phi_1]$	PRISM, PLASMA-Lab, MRMC and MC2
Release	Within the $\bowtie p$ bounds, ϕ_2 holds continuously until ϕ_1 holds, namely ϕ_1 releases ϕ_2 .	$P_{\bowtie p}[\phi_1 \text{ R } \phi_2]$ or $P_{\bowtie(1-p)}[\neg \phi_1 \text{ U } \neg \phi_2]$	PRISM, PLASMA-Lab, Ymer, MRMC and MC2
Weak Until	Within the $\bowtie p$ bounds, ϕ_1 holds continuously until ϕ_2 holds, if ϕ_2 does not hold, then ϕ_1 holds forever.	$P_{\bowtie p}[\phi_1 \text{ W } \phi_2]$ or $P_{\bowtie(1-p)}[\neg \phi_2 \text{ U } (\neg \phi_1 \ \& \ \neg \phi_2)]$	PRISM, PLASMA-Lab, Ymer, MRMC and MC2

Key. ϕ_1 , and ϕ_2 are state formulas; \bowtie is a relation of $\{<, >, \leq, \geq\}$; p is probability $\in [0, 1]$; and \bowtie is negation of the inequalities.

3 Experiments

In order to identify the performance of SMC tools, we verified instances of the 11 patterns in Table 1 on 675 up-to-date (August 2016) biological models taken from the BioModels database (39) in SBML format. In order to focus on the model structure analysis, they have fixed stochastic rate constants of all reactions to 1.0 and the amounts of all species to 100. Sanassy et al. (19) have previously (2015) modified and considered 465 of these models in a similar fashion. We have made same changes to the 210 new models considered here. The models tested ranged in size from 2 species and 1 reaction, to 2631 species and 2824 reactions. Figure 1 shows the distribution of model sizes, where we take “size” to be the product of species count and reaction count. All experiments were conducted on the same desktop computer (Intel i7-2600 CPU @ 3.40GHz 8 cores, 16GB RAM, running under Ubuntu 14.04).

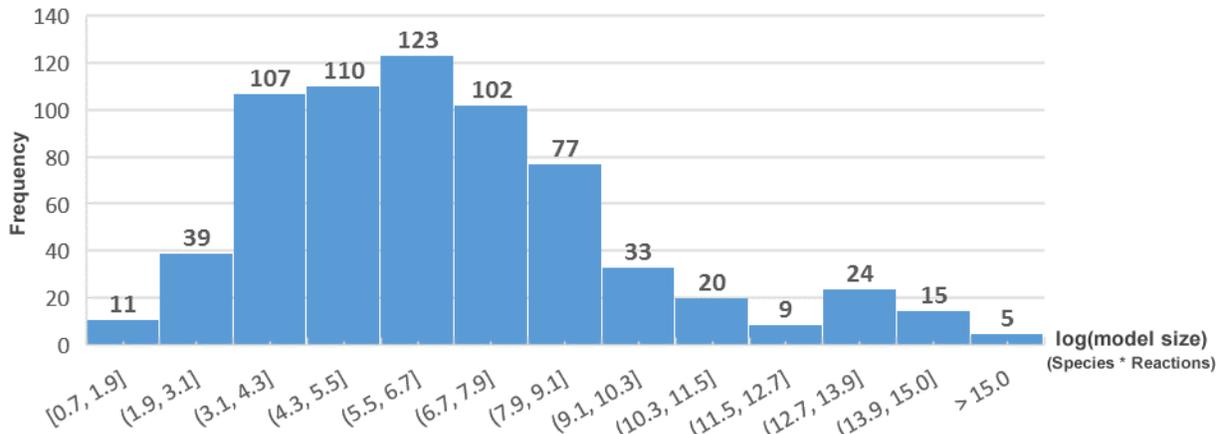


Figure 1: Model size distribution. The X-axis plots the (natural logarithm of) model size and the Y-axis represents the frequency of models within the corresponding X-axis interval.

3.1 Feature Selection

Network analysis of biological systems is a principal component of systems biology which has been used to identify complex molecular interactions of the biological systems (40). A *graph* consists of vertices and edges. Typically, a *vertex* signifies an entity and an *edge* represents the existence of relationship between two entities. In *undirected graphs*, edges do not have direction. Thus, the relationship is symmetric. In *directed graphs*, each edge has a direction starting from one vertex and pointing to itself or another vertex. One of the basic features of a vertex is its *degree* which is the number of edges linked to it. In directed graphs, the number of edges pointing to a vertex is called *in-degree* and the number of outgoing edges is called *out-degree*. Graph *density* is the ratio between the number of existing edges and the total number of possible edges (40, 41). Essentially, it measures how sparse or dense the vertices are interconnected. A *connected component* is the maximal subset of vertices in an undirected graph, where any two vertices are connected by one or more edges (40). More specifically, *strongly connected components* take the direction of the edges into account, whereas the *weakly connected components* ignore the directions (40, 42). A *vertex cut* or *articulation point* is a vertex in a connected component of a graph whose removal causes the subgraphs becoming separated (43). A *biconnected component* is a subgraph which does

not have articulation points (43). *Reciprocity* in directed graphs measures the proportion of mutual connections, i.e. reciprocal edges (40).

The verification time expended by SMC tools primarily depends both on the characteristics of the model, such as the number of species and reactions, and on the property being queried (5, 19), and it is therefore crucial to identify those model features which effectively discriminate between different SMC tools’ performance characteristics.

Sanassy et al. (19) have previously used topological features from the model’s network to assist in SSA performance prediction. They represented models using species and reaction dependency graphs, then used various topological features of these graphs to predict the fastest SSA among nine algorithms. Initially, they considered 109 features of species and reaction graph, and they were able to successfully predict the fastest SSA for 380 models with 65% accuracy. Computing some of these features are computationally demanding and, therefore, they excluded some of more expensive features and considered only 16 relatively less expensive features for each graph type, 32 in total, for their second experiment. The final set of features they used are shown in first column of Table 2. Decreasing the number of features slightly reduced their accuracy of prediction, to 63%.

In our experiments we aim to increase predictive power without compromising on computation time of the feature extraction. In addition to the graph topological features, we considered twelve new features which mostly do not require graph construction – these are listed in the second column of Table 2. The bar chart in Figure 2 shows the average computational time (in nanoseconds) required when using each topological feature. As the figure shows, the new topological features are computationally inexpensive. In order to identify which of the properties are most important for our purposes, we conducted feature selection analysis with the *feature importance algorithm* of Extremely randomized trees (44–47). The data points on the line graph in Figure 2 show the ‘percentage importance’ of each feature. The results show that graph theoretical features like reciprocity, weakly connected components, biconnected components and articulation points are computationally expensive but

Table 2: Topological features of models.

Graph Features used by Sanassy et al. (19)	New Features (reported here)
Number of Vertices	Number of non-constant species (regular species whose populations can change, e.g. catalysts cannot be in this category.)
Number of Edges	(Species * Reactions) (number of species multiplied by number of reactions)
Density of Graph	Update values (min/mean/max and total number of variable changes when reactions trigger)
Degrees (min/mean/max of incoming, outgoing and all edges)	Sum of the Degrees (total number of incoming, outgoing and all edges, for each graph)
Weakly connected components	
Articulation points	
Biconnected Components	
Reciprocity of Graph	

Key. The term “update values” refers to the number of species whose populations change when a reaction triggers. For example, if the reaction " $2A + 3B \rightarrow C$ " triggers, the number of variables updated is 3, namely A , B and C .

actually contribute less to predictive power than the computationally less expensive features. Hence, we removed these relatively expensive features, and the final properties set used in our work consists only of computational inexpensive features.

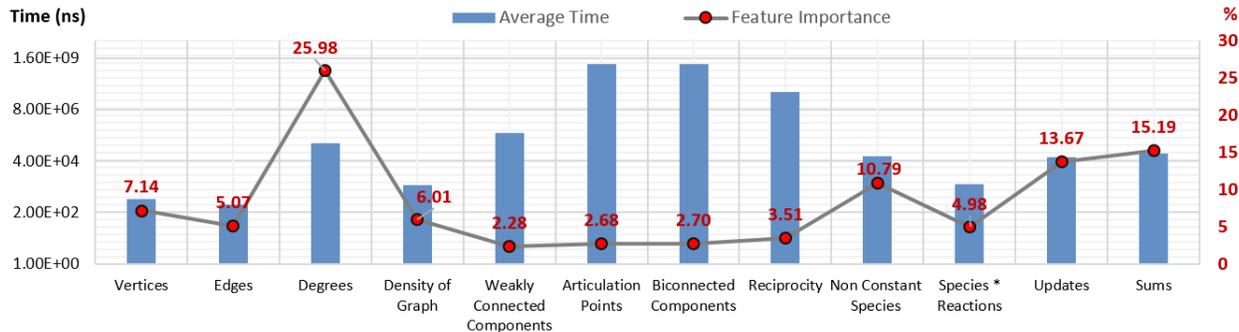


Figure 2: Average computational time and feature importance associated with various topological properties of models.

We performed a sanity check to show the proposed features are important for the prediction and the expensive features are not contributing much to the prediction, by repeating

the experiments reported by Sanassy et al. (19), but using three different feature groups. The first group consists of the 32 features proposed by Sanassy et al., the second group added in the 12 new features we proposed (44 features in total), and the third group is the same as the second group but excludes the computationally expensive graph-theoretic features (reciprocity, weakly connected components, biconnected components and articulation points)—this third group consists of 36 features. Each of the feature groups of 380 models were submitted to a linear SVC classifier for predicting the fastest *Stochastic Simulation Algorithm* among nine algorithms. The prediction accuracies of 10-fold cross-validation for each group are shown in Table 3. Using the features of group 2 achieved the highest prediction accuracy (69.7%), but as we already stated these group includes some relatively computationally expensive properties; on removing these (group 3), the prediction accuracy dropped just 1% relative to group 2, but the feature computation time was considerably reduced. Therefore, to make the automated prediction for large system tractable, we used the features of group 3 for the rest of experiments reported in this study. The identified feature group is provided to the machine learning algorithms which is explained in later sections.

Table 3: Accuracies of the three different topological feature groups for predicting SSAs.

	Group 1	Group 2	Group 3
Accuracy	63%	69.7%	68.6%

Key. Please note that the results shown here are the fastest Stochastic Simulation Algorithm predictions, the fastest Statistical Model Checking tool predictions are reported in Section 3.5.

3.2 Performance Benchmarking of SMC Tools

In our previous study (5), we benchmarked the performance of five SMC tools for verification of 465 models against 5 patterns. Typically, increasing the sample size provides more reliable predictions with machine learning algorithms. Therefore, we increased the number of analysed biological models to include all of the up-to-date stochastic SBML formatted models available in the BioModels Database (39). Eventually, we tested each of the five tools

against the 675 models in SBML format. Each tool/model pair was tested against the 11 property patterns. We developed a tool for translating the SBML models and pattern-based properties according to each SMC’s specification requirements. For each test the number of simulation traces was set to 500, and the number of steps for each trace was set to 5000.

Each test was repeated three times and their average time is considered. The elapsed time for each run includes the time required for model parsing, simulation and verification times, and where one tool depends on the use of another then the execution time of the external tool is included in the total execution time. In particular, the total times reported for MRMC and MC2 are not their execution times only, but also include the time consumed for generating the transition matrices and simulation traces, using PRISM and Gillespie2, respectively. In addition, when the external tool failed to produce the required inputs for the SMC tool, then we consider that the model cannot be verified by the SMC in question. In order to keep the experiment tractable, we set the upper limit for execution time to one hour, so if a tool fails to verify the model within the time limit, we say the model cannot be verified by the tool.

Table 4: The number of models verified with different property patterns by each SMC tool.

PATTERNS	PRISM		PLASMA-Lab		Ymer		MRMC		MC2	
	Verified	Fastest	Verified	Fastest	Verified	Fastest	Verified	Fastest	Verified	Fastest
Eventually	364	18	675	248	644	402	116	3	668	4
Always	480	80	675	132	644	457	118	2	668	4
Follows	N/A	N/A	675	575	N/A	N/A	116	39	664	61
Precedes	672	170	675	18	644	486	113	0	664	1
Never	542	103	675	147	644	422	116	1	668	2
Steady State	N/A	N/A	675	579	N/A	N/A	80	30	668	66
Until	592	125	675	82	644	465	112	0	664	3
Infinitely Often	N/A	N/A	675	604	N/A	N/A	N/A	N/A	668	71
Next	658	581	675	17	N/A	N/A	118	36	675	41
Release	622	151	675	49	644	472	111	0	664	3
Weak Until	591	126	675	82	644	465	112	0	664	2

Key. “Verified” columns represent the number of models could be verified by each tool, and “Fastest” columns show the number of models for which the corresponding tool verified them as the fastest tool. The patterns which are not supported by corresponding tools are marked as Not Applicable (N/A).

Table 4 summarises the experiment results. The “verified” column for each tool shows the

number of models that could be verified by the corresponding tool against each property pattern. PLASMA-Lab could verify all models, MC2 could verify most models for all property patterns, except for *Precedes*, which PRISM could verify more often than MC2. MC2 failed to verify only a few models within the available time. Ymer also could verify most of the models, but couldn't handle and repeatedly crashed for 31 large models. PRISM's capacity for verification depends on the pattern type, for example, it could verify only 364 models against the *Eventually* pattern but it could verify almost all models, 672, for the *Precedes* pattern. We believe that PRISM fails because in order to have a reliable approximation, it requires a greater simulation depth for unbounded property verification. MRMC could verify fewer models than the other SMCs for all property patterns, because it relies on PRISM for transition matrix generation. However, for medium sized and large models PRISM failed to build and export the transition matrices – we believe this was due to a CU Decision Diagram (CUDD) library crash.

The “Fastest” columns in Table 4 show the number of models for which the corresponding tool verified them as the fastest tool. In addition, Figure 5 illustrates the relationship between fastest tool and model size. Ymer could verify most of the models fastest (for the supported property patterns), however, as Figure 5 shows, it was generally fastest for relatively small sized models. PRISM and PLASMA-Lab are generally fastest for medium to large sized models. The number of models verified fastest by PRISM and PLASMA-Lab dramatically change based on the verified property pattern. MRMC and MC2 are the fastest tools for fewer models and they perform best only for small sized models. They do slightly better for the *Follows*, *Steady State* and *Infinitely Often* patterns where they compete with fewer tools.

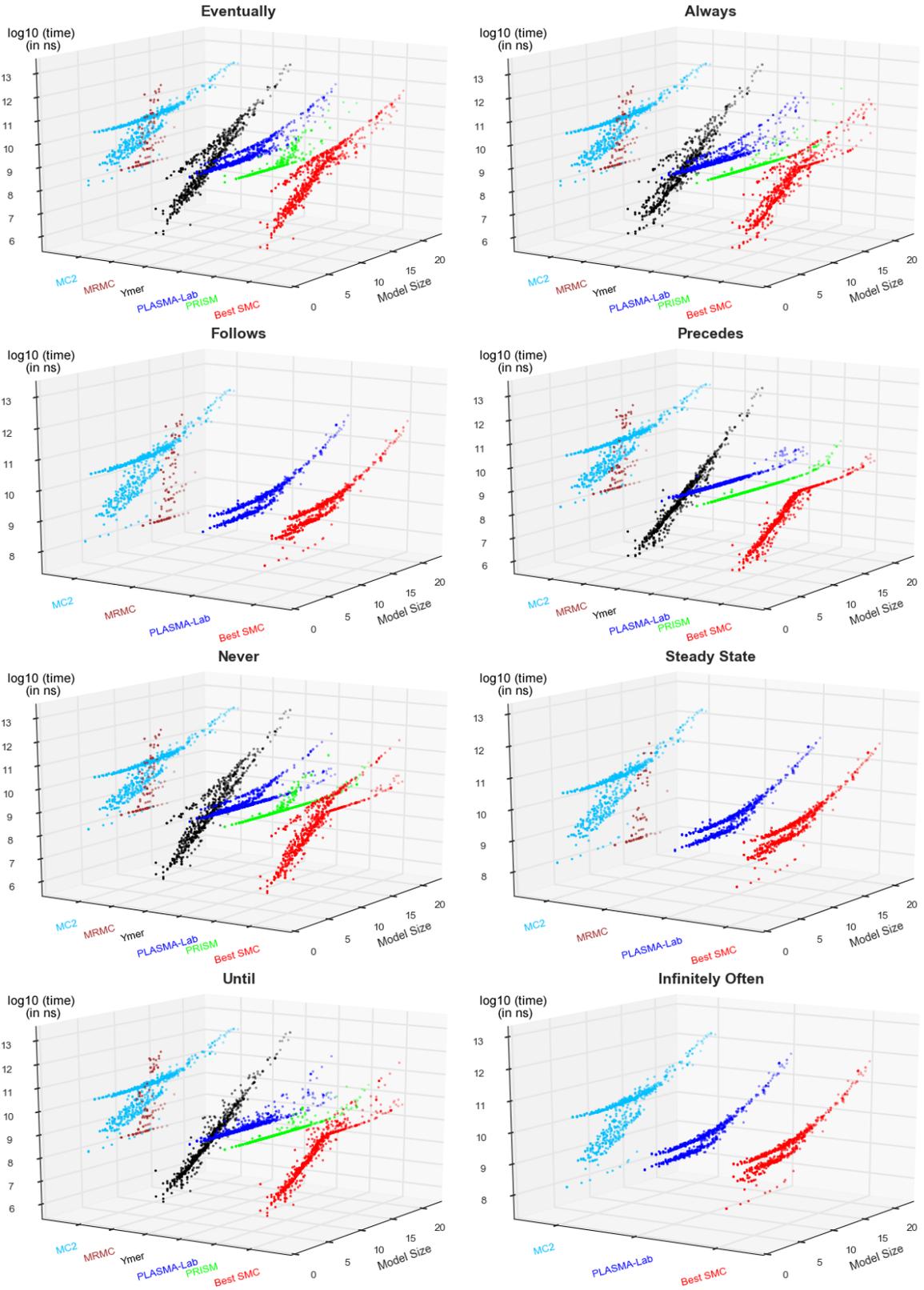


Figure 3: Performance comparison of each tool against the best performance for verification of property patterns. Cont.

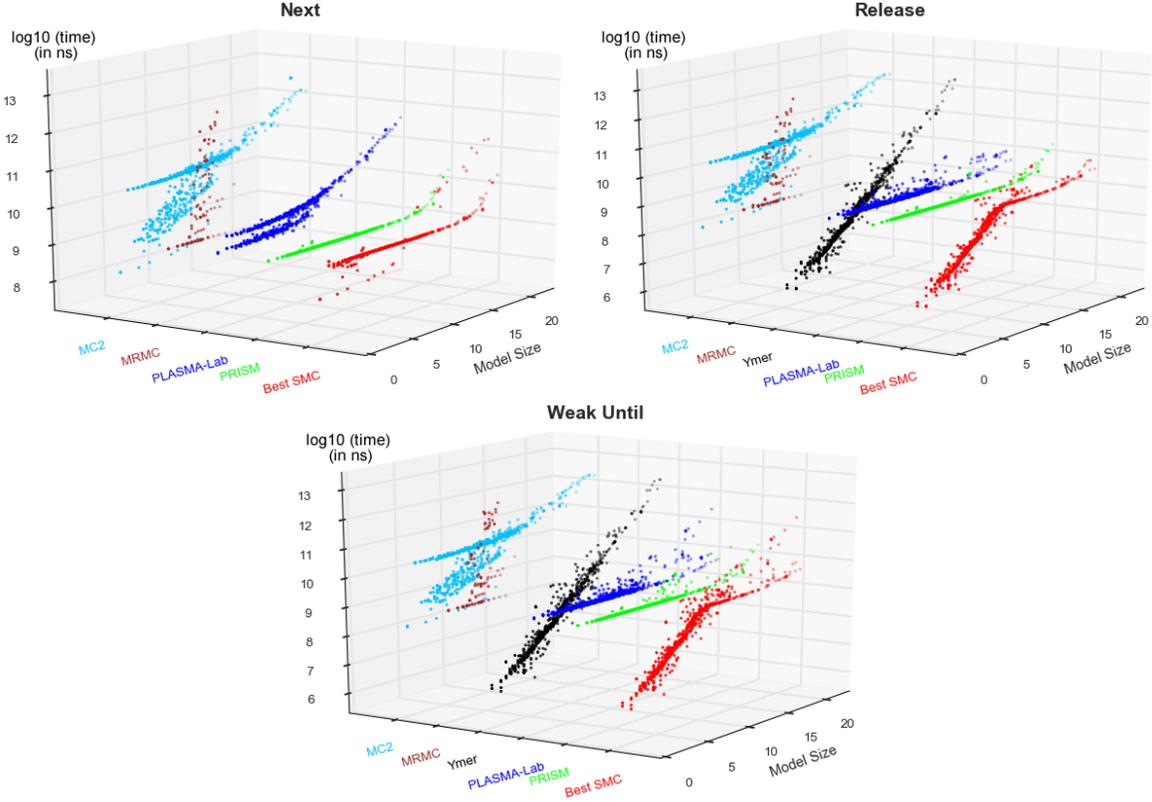


Figure 4: Performance comparison of each tool against the best performance for verification of property patterns.

Key. For Figure 3 and Figure 4, the property patterns are shown on each panel, X-axes represent the model size (species*reactions) on logarithmic scale (\log_2), and Y-axes are the SMC tools compared with the fastest tool. Z-axis is the logarithmic scale (\log_{10}) of consumed time in nanoseconds.

Figure 3 and Figure 4 provide complementary information to Table 4 and Figure 5. They illustrate the verification time of each tool, alongside with the fastest option, against the model size. MC2 and MRMC support most of the patterns, however, their verification time is generally higher than the other tools. Especially, MRMC can verify a few small models and for the larger models, its verification time increases exponentially. The verification time of Ymer roughly increases linearly. Hence, for the small sized models, the verification time is short and it steadily increases when the model size increases. Consequently, Ymer usually performs better for small sized models. For the Follows, Stead-State and Infinitely-Often patterns, PLASMA-Lab verification time displays approximately an exponential growth, but it is still the fastest one as it competes only either with MC2 or both MRMC and MC2.

PLASMA-Lab has a similar verification time curve for the Next pattern too. However, here it also competes with PRISM, and PRISM mostly presents a better performance for the Next pattern. PRISM performances for the verification of different model sizes are usually close to each other. Therefore, like PLASMA-Lab PRISM also is not the fastest option for small sized models, and it is often better for the larger models.

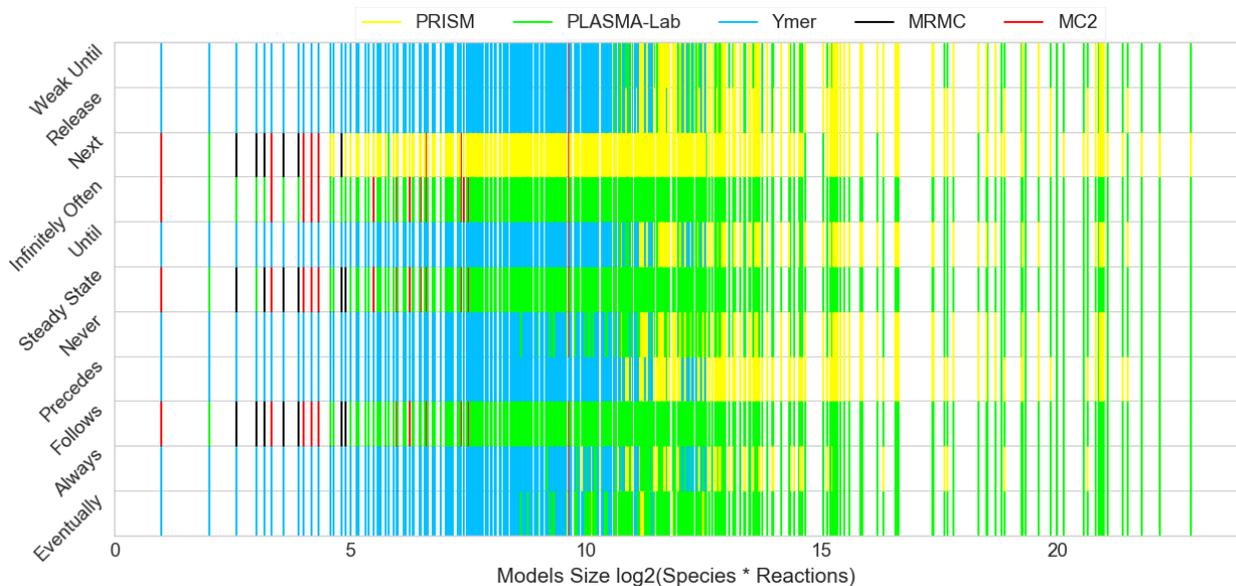


Figure 5: Fastest SMC tools for verifying each model against each property pattern.

Key. X-axis represents logarithmic scale of model size and Y-axis shows the property patterns. For each model (log-of-size shown on the X-axis) a one-unit vertical line is drawn against each pattern (shown on the Y-axis), where the line's colour shows the fastest SMC. The distribution of the fastest tools are shown in Figure 6.

The results show that the performance of tools significantly changes based on models and property patterns, which makes it extremely difficult to know which tool is best without the assistance of an automated system.

3.3 Automating SMC Tool Prediction

A *machine learning algorithm* (aka ‘*learner*’, ‘*classifier*’) receives a set of samples where each sample consists of an ordered pair of a *vector of features*, called *input* (or *independent variables*), and a variable, called *output* (or *target variable* or *dependent variable*). It starts

with some initial parameters, and it *learns* by optimising its parameters based on the samples (48), this is called *training* the algorithm. A *learned*, i.e. *trained*, algorithm is basically a function, f , that maps an input to a target variable. After having the function, f , it can be used for getting the output variables of some new inputs, this process is called *prediction*. In our study, the machine learning algorithms used a set of 675 samples, which have the model features (identified in the ‘Feature Selection’ section) as inputs, and the fastest SMC (identified in the ‘Performance Benchmarking of SMC Tools’ section) as the target variables.

We used 5 machine learning techniques and 2 random selection algorithms (49) for predicting the fastest SMC tool. The random selection algorithms, referred to here as Random Dummy (RD) and Stratified Dummy (SD), were used for comparing the success rate of each algorithm with random prediction. The RD classifier ‘guesses’ the SMC tools blindly, that is, with probability $1/5$ picks one of the 5 verification tools at random., whereas the SD classifier knows the distribution of the fastest SMC tools that is shown in Figure 6, as it is inferred from Table 4. The RD classifier acts as a proxy for the behaviour of the researchers who do not know much about model checking tools, while SD can be considered as mirroring the behaviour of experienced verification researchers who know the patterns supported by each tool and fastest tools distribution but do not know which tool is best for a specific property to be checked on *a specific model*. The remaining five methods are; a support vector classifier (SVM) (50), logistic regression(LR) (51), a nearest neighbour classifier (KNN) (52) and two types of ensemble method, namely, Extremely Randomized Trees (ERT) (44) and Random Forests(RF) (53) (despite their name these are not random classifiers but ensemble classifiers). We used the scikit-learn library (54, 55) implementation of these classifiers in our experiments.

We considered three different accuracy scores in our experiments. The first score, ‘ $S1$ ’, is the regular accuracy score with *10-fold cross-validation*. *Accuracy* is the percentage of correct estimation of the fastest SMC tool over the sample size (the ratio of true positive prediction over the sample size). *10-fold cross-validation* consists of 10 consecutive rounds.

Initially, all data is divided into 10 blocks, and 9 blocks are used for training the machine learning algorithm, and one block is used for testing the accuracy of the algorithm that is trained over the 9 blocks. Then the testing data is put back to the data set. In the successive rounds, another block is used as the testing data, and the rest 9 of the blocks are used for training. This procedure is repeated until every block used exactly one time for testing. The average accuracy of all rounds is regarded as the final accuracy of the machine learning algorithm.

The second score, ‘S2’, is calculated by putting a threshold bound for considering a prediction to be correct, namely, in this case if the time difference of the actual fastest SMC tool and the predicted SMC tool is not more than 10% of the actual fastest SMC tool time, then we regard the prediction as correct. In the third score, ‘S3’, instead of time difference, we set the threshold as the order of the fastest SMC tool. That is, if the predicted SMC tool is the second fastest tool, then we regard it as the correct prediction.

The experiment results with the first score of each classifier for different property patterns are shown in Figure 7 and their accuracy values are tabulated in Table 5. The success rates of all classifiers were higher than the random classifiers. ERT was the most frequent winner, as it had best prediction accuracy for 6 patterns (for *Infinitely Often*, ERT and LR both have highest accuracy, 95%), whereas the SVM classifier was the second best winner with highest prediction accuracy for 5 patterns, ERT and SVM are here after referred to as the best classifiers. The accuracies of prediction of the best classifiers were over 90% percent for all pattern types. We also want to show whether predicting different property patterns with the same classifier is statistically significant. Therefore, we measured the P-values of each classifiers across different property patterns, by comparing the accuracy scores of cross-validation of each classifiers with the Friedman test (56) of Python SciPy library (57), which are shown in the last column of Table 5. The result clearly shows same classifier for different patterns are statistically significant, therefore it would not be the best practice to use just one classifier type for all pattern types. We also measured the P-values of different classifiers

for predicting same property pattern, which are shown in the last row of the Table 5. The low P-values also show that different methods are statistically different.

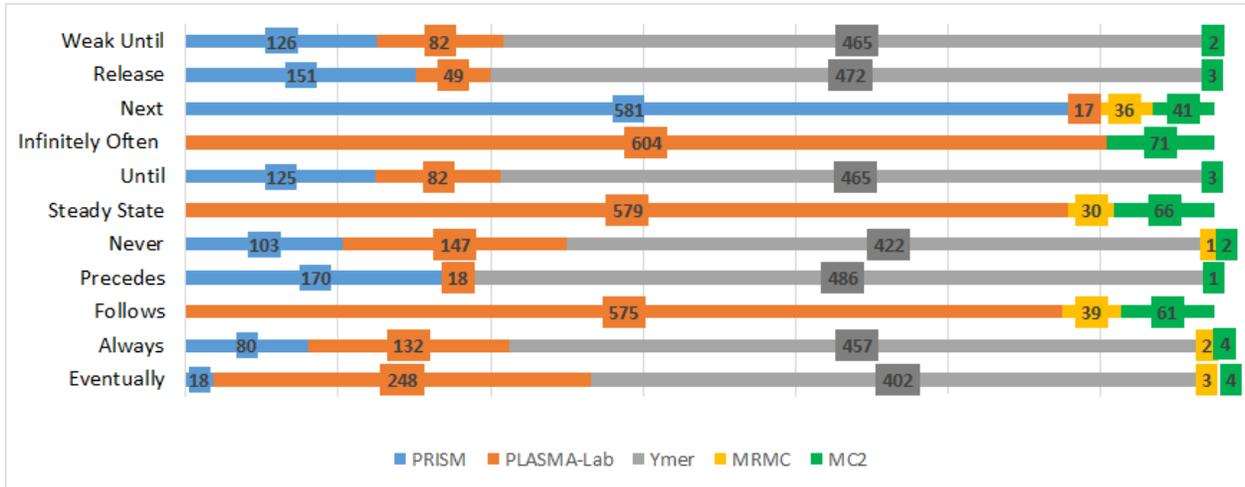


Figure 6: Distribution of fastest SMC tools for different property patterns.

Key. Distribution of the fastest SMC tools used by Stratified Dummy (SD) classifier. Values in boxes show the number of models were verified by the corresponding SMC as being the fastest tool.

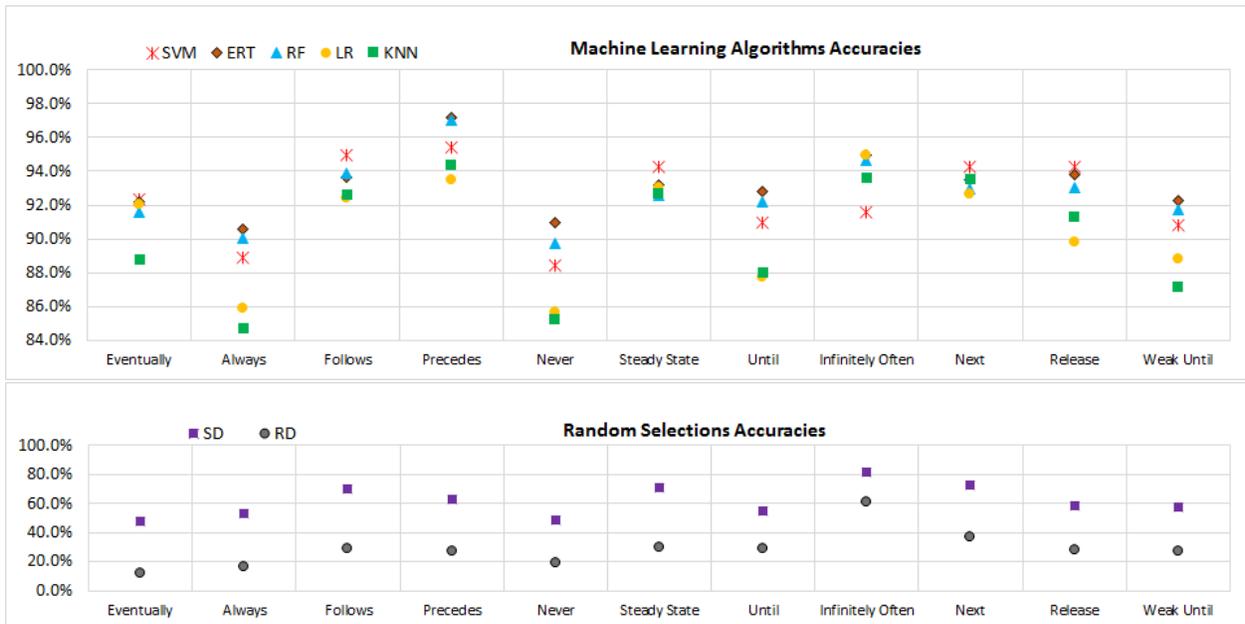


Figure 7: Accuracies with first score for the fastest SMC prediction with different algorithms.

Table 6 shows the experiment results with different score settings. ‘S2’ rows report the experiments using the second score settings, and ‘S3’ rows report the third score. The accuracy of S2 experiment settings is not much higher than the first score which regarded only

Table 5: Accuracy values with first score.

	Eventually	Always	Follows	Precedes	Never	Steady State	Until	Infinitely Often	Next	Release	Weak Until	P-value
SVM	92.4%	88.9%	95.0%	95.4%	88.5%	94.2%	91.0%	91.6%	94.3%	94.2%	90.8%	4.10E-08
ERT	92.2%	90.5%	93.6%	97.2%	91.0%	93.2%	92.8%	95.0%	93.5%	93.8%	92.3%	8.76E-06
RF	91.6%	90.1%	93.9%	97.0%	89.8%	92.6%	92.2%	94.7%	92.9%	93.1%	91.7%	3.27E-05
LR	92.0%	85.9%	92.4%	93.5%	85.6%	93.0%	87.8%	95.0%	92.6%	89.8%	88.8%	1.39E-07
KNN	88.8%	84.7%	92.6%	94.4%	85.2%	92.7%	88.0%	93.6%	93.5%	91.3%	87.1%	2.81E-08
SD	47.7%	53.4%	70.5%	63.3%	48.8%	70.7%	54.7%	81.5%	72.9%	58.8%	57.4%	2.70E-15
RD	12.4%	16.3%	29.1%	27.4%	19.1%	29.6%	28.7%	61.2%	36.9%	28.4%	27.3%	6.62E-13
P-value	1.28E-08	4.63E-09	8.19E-08	4.75E-04	2.58E-03	2.46E-07	2.61E-04	3.10E-08	9.12E-08	2.82E-08	1.13E-07	

the actual fastest tool prediction as correct, but the accuracy of S3 is significantly higher, because it ‘lumps together’ the fastest and second fastest tools, but the time differences between the second best and the actual best tool can be orders of magnitude, i.e. much more than 10%. For the Follows, Steady State and Infinitely Often patterns, the accuracies of SD and RD are relatively better under these more relaxed scores, because there are fewer tools which support these patterns, hence they have higher chance of correct prediction. The P-values of ‘S2’ and ‘S3’ scores are low, like ‘S1’, which shows that different classifiers should be used for different property patterns, namely single classifier cannot be the best for all pattern types.

Table 6: The prediction accuracy with different score settings.

		Eventually	Always	Follows	Precedes	Never	Steady State	Until	Infinitely Often	Next	Release	Weak Until	P-value
SVM	S2	94.1%	91.1%	95.7%	96.6%	89.8%	95.3%	91.9%	92.3%	95.4%	95.4%	92.6%	1.1E-07
	S3	98.7%	96.4%	99.1%	98.1%	94.4%	99.3%	95.1%	100.0%	97.2%	97.9%	96.7%	4.1E-08
ERT	S2	93.7%	92.2%	94.8%	98.1%	92.7%	94.1%	94.3%	95.9%	94.7%	95.3%	93.9%	2.0E-04
	S3	98.4%	96.9%	98.5%	99.6%	96.1%	99.0%	97.8%	100.0%	96.6%	97.8%	97.6%	2.0E-06
RF	S2	93.4%	91.8%	95.0%	98.7%	91.5%	93.6%	93.6%	95.4%	94.3%	95.4%	93.2%	2.3E-05
	S3	99.0%	97.0%	99.3%	99.9%	96.0%	99.1%	97.3%	100.0%	96.3%	97.6%	97.0%	2.3E-08
LR	S2	93.8%	88.6%	93.6%	95.1%	87.5%	94.2%	90.0%	95.9%	93.8%	91.9%	90.4%	2.0E-07
	S3	99.0%	95.4%	99.1%	97.3%	93.8%	99.7%	95.1%	100.0%	96.1%	95.1%	95.1%	7.3E-10
KNN	S2	90.2%	87.5%	93.6%	96.0%	87.7%	93.6%	90.5%	94.5%	94.7%	92.8%	89.6%	7.0E-08
	S3	96.7%	94.5%	98.1%	98.7%	93.2%	98.5%	95.4%	100.0%	96.0%	95.1%	94.4%	3.2E-09
SD	S2	49.3%	55.7%	70.5%	64.9%	52.3%	71.4%	56.9%	82.4%	78.1%	60.7%	59.4%	2.7E-15
	S3	71.7%	71.9%	90.5%	75.0%	72.4%	90.4%	70.8%	100.0%	86.8%	72.4%	74.8%	1.3E-12
RD	S2	13.6%	17.3%	30.5%	31.8%	19.7%	30.7%	30.9%	61.5%	38.4%	31.7%	28.9%	1.6E-13
	S3	32.9%	35.0%	65.0%	49.5%	39.4%	64.3%	48.0%	99.3%	56.0%	47.6%	47.9%	8.2E-15
P-value	S2	1.5E-08	2.7E-08	7.4E-08	8.7E-10	9.8E-09	2.2E-07	4.5E-09	2.0E-08	9.3E-08	7.3E-08	5.5E-08	
	S3	3.6E-09	1.8E-07	4.0E-08	2.6E-09	4.6E-08	5.1E-08	1.2E-08	6.2E-03	3.6E-08	1.6E-08	2.0E-08	

Key. S2 and S3 rows represent the scores.

3.4 Performance Gain and Loss

For our final experiments we assessed the performance gain and loss associated with using the classifier with the best prediction (either ERT or SVM) versus the random classifiers for each property pattern. The performance gain is the time saved by using the best classifier predictions instead of random choices. In this regard, performance gain is the time difference between the verification time of the tools predicted by the best classifier and the random classifiers.

Figure 8 shows the total time consumption for verifying all models with the actual fastest SMC tools, the best classifier predictions, and the random classifiers predictions. When a classifier predicted a tool which cannot actually verify the model, then we declared its execution time to be the experimental upper time limit, namely 1 hour. The time difference (the performance gain) between the best classifiers and the SD classifier is: minimum 63 with Infinitely Often pattern, maximum 3002 with Always pattern, and average 1848 minutes for all patterns. The time difference between the best classifiers and RD is even larger: min 528, max 12506 and average 6109 minutes for all patterns. The results show that by using the best classifier predictions allows a significant amount of time can be saved – up to 208 hours!

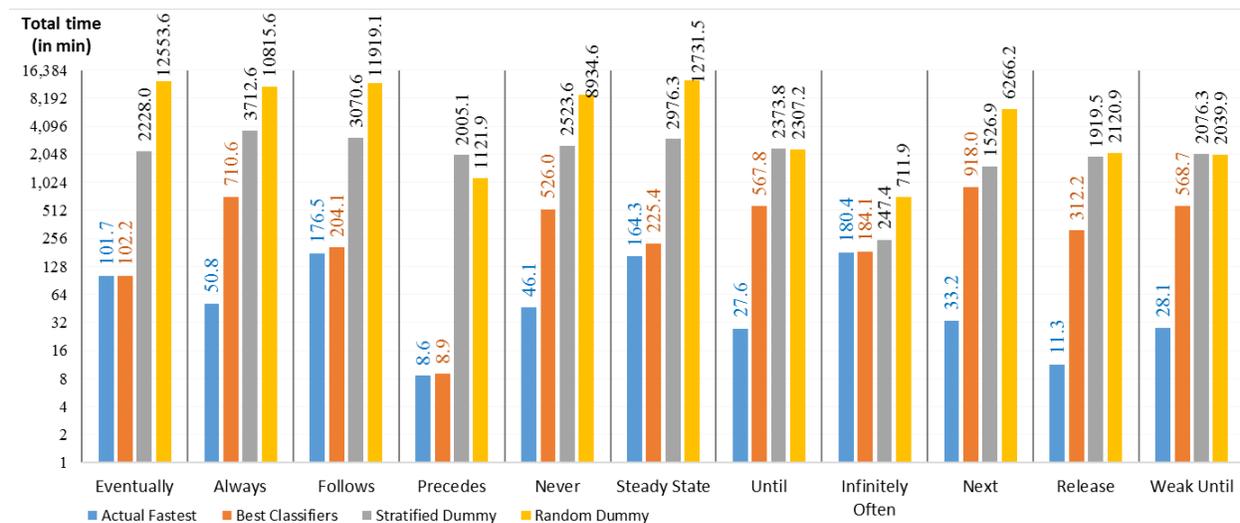


Figure 8: Total time consumed for verifying all models.

Generally, the outcomes of mispredictions can be as important as the correct predictions. In this regard, we measured the performance loss caused by each inaccurate prediction with the best classifiers and the random classifiers. Figure 9 shows the time difference between the actual best SMC tools and the predicted SMC tools of inaccurate predictions. The minimum performance loss of the best classifier is min 0.3 minutes with Precedes, max 885 with Next, and average 318 minutes for all patterns. Similarly, the performance loss for SD: min 67, max 3662 and average 2167 minutes; for RD: min 532, max 12567, and average 6427 minutes. As the results suggest the performance losses of the best classifiers are always less than the random classifiers' loss. More specifically SD and RD causes performance loss 7, and 20 times, respectively, more than the best classifiers.

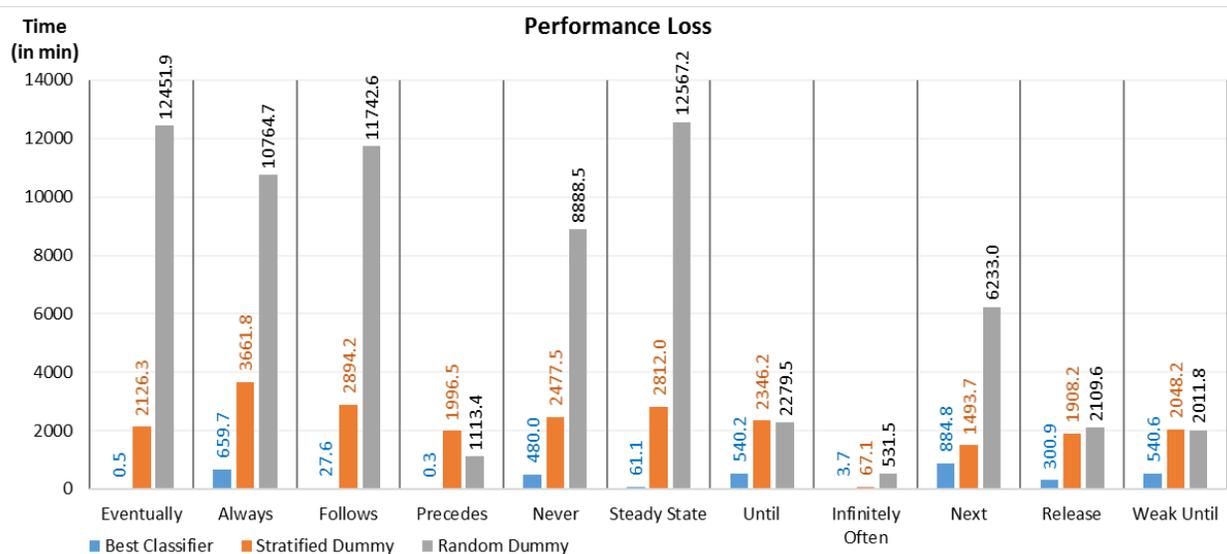


Figure 9: The mean of performance loss when the best classifiers mispredicts.

3.5 SMC Predictor Tool

We developed a software utility tool, SMC Predictor, which accepts an SMBL model and a property pattern file as input. The property pattern file can include more than one pattern queries. SMC predictor returns the fastest SMC tool prediction for each pattern. The tool architecture and work-flow is show in Figure 10 . The tool modifies the received input model

to focus on model feature analysis. This is done by fixing the stochastic rate constant and the number of species to 1 and 100, respectively, and removing multiple compartments. The modified model is passed to the Model Topology Analysis component which extracts the graph features of species and reactions graphs, such as edges, degrees, and the non-graph features, such as number of updates (*see* Section “Feature Selection”). The model features together with property pattern are delivered to the Predictor component. The Predictor component initializes the best classifier based on the property pattern, and conveys the model features to the classifiers to predict the fastest SMC tool, then returns the prediction result to the user. The grammar of property patterns, experimental data, the tool, its usage and requirements, and some exemplary models are available at www.smcpredictor.com.

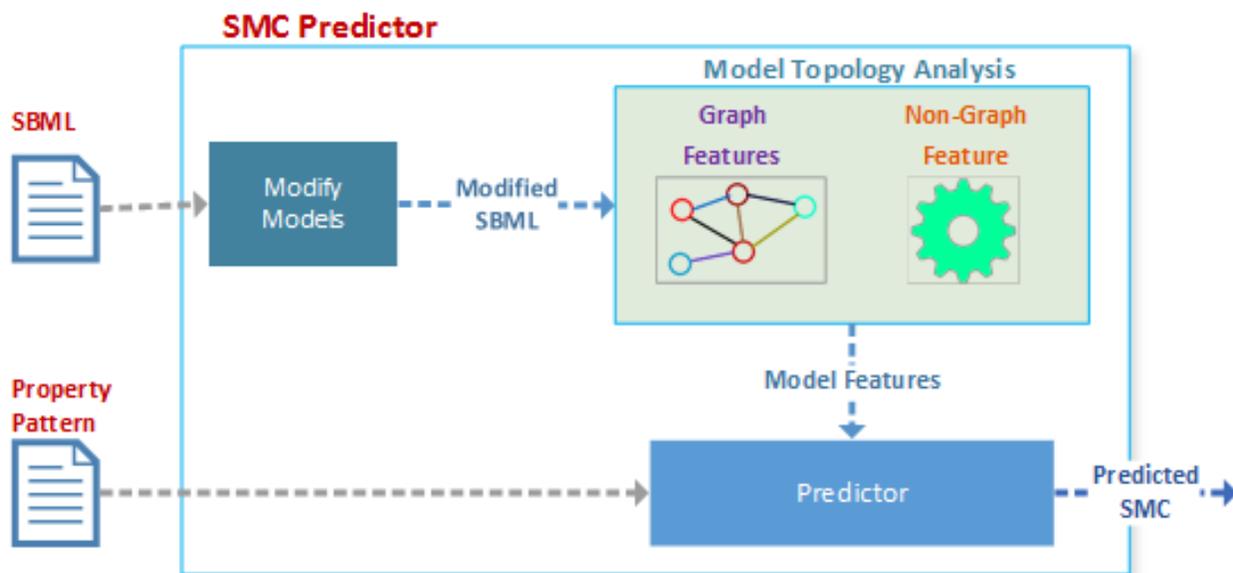


Figure 10: SMC predictor architecture and work-flow.

4 Conclusion

In this paper, we have proposed and implemented a methodology to automatically predict the most efficient SMC tool for any given model and property pattern. We have proposed a set of model features which can be used for SMC prediction, reduce the computation time

required, and increase the prediction accuracy. In this paper, we have also extended our previous work (5) on performance benchmarking by verifying 675 biological models against 11 property patterns. In addition, in order to match the fastest SMC tool to a given model, we have proposed a new set of model features. We have demonstrated that the proposed features are computationally cheap and important for the predictive accuracy. Using several machine learning algorithms we could successfully predict the fastest SMC tool with over 90% accuracy for all pattern types. Finally, we have demonstrated that by using automated prediction, users can save a significant amount of time. For the next stage of our work, we are aiming to integrate the automated fastest SMC prediction process into some of the larger biological model analysis suites, for example KPWORKBENCH (14) and INFOBIOTICS WORKBENCH (17).

Acknowledgement

NK, MG and SK acknowledge the EPSRC for grant EP/I031642/2.

NK also acknowledges grants EP/J004111/2, EP/L001489/2 & EP/N031962/1

MEB is supported by a PhD studentship provided by the Turkey Ministry of Education.

References

1. Fisher, J., and Henzinger, T. A. (2007) Executable cell biology. *Nat Biotech* 25, 1239–1249.
2. Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*; MIT Press, 1999.
3. Younes, H. L. S., and Simmons, R. G. *Proceedings of 14th International Conference on Computer Aided Verification*; Springer, 2002; pp 223–235.
4. Huth, M., and Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd ed.; Cambridge University Press, 2004.

5. Bakir, M. E., Gheorghe, M., Konur, S., and Stannett, M. *Membrane Computing*; Lecture Notes in Computer Science; Springer, 2017; pp 119–135.
6. Konur, S., Gheorghe, M., Dragomir, C., Mierla, L., Ipate, F., and Krasnogor, N. (2015) Qualitative and Quantitative Analysis of Systems and Synthetic Biology Constructs using P Systems. *ACS Synthetic Biology* 4, 83–92.
7. Frisco, P., Gheorghe, M., and Pérez-Jiménez, M. J., Eds. *Applications of Membrane Computing in Systems and Synthetic Biology*; Emergence, Complexity and Computation; Springer International Publishing, 2014; Vol. 7.
8. Konur, S., Gheorghe, M., Dragomir, C., Ipate, F., and Krasnogor, N. (2014) Conventional Verification for Unconventional Computing: A Genetic XOR Gate Example. *Fundam. Inf.* 134, 97–110.
9. Sanassy, D., Fellermann, H., Krasnogor, N., Konur, S., Mierlă, L., Gheorghe, M., Ladroue, C., and Kalvala, S. Modelling and stochastic simulation of synthetic biological Boolean gates. 16th IEEE International Conference on High Performance Computing and Communications. 2014; pp 404–408.
10. Khalis, Z., Comet, J. P., Richard, A., and Bernot, G. (2009) The SMBioNet method for discovering models of gene regulatory networks. *Genes, Genomes and Genomics* 3, 15–22.
11. Faeder, J. R., Blinov, M. L., and Hlavacek, W. S. *Methods in Molecular Biology, Sys. Bio.*; Methods in Molecular Biology; Humana Press, 2009; Vol. 500; pp 113–167.
12. Ciocchetta, F., and Hillston, J. (2009) Bio-pepa: A framework for the modelling and analysis of biological systems. *Theoretical Computer Science* 410, 3065–3084.
13. Batt, G., Besson, B., Ciron, P.-E., Jong, H., Dumas, E., Geiselmann, J., Monte, R.,

- Monteiro, P., Page, M., Rechenmann, F., and Ropers, D. *Bacterial Molecular Networks*; Methods in Molecular Biology; Springer New York, 2012; Vol. 804; pp 439–462.
14. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., and Mierla, L. *Membrane Computing*; Lecture Notes in Computer Science; Springer Berlin Heidelberg, 2014; Vol. 8340; pp 151–172.
 15. Bakir, M. E., Ipate, F., Konur, S., Mierla, L., and Niculescu, I. In *Membrane Computing*; Gheorghe, M., Rozenberg, G., Salomaa, A., Sosik, P., and Zandron, C., Eds.; Lecture Notes in Computer Science; Springer, 2014; pp 158–178.
 16. kPWorkbench. <http://kpworkbench.org/>, n.d.; [Online; accessed 25/07/17].
 17. Blakes, J., Twycross, J., Romero-Campero, F. J., and Krasnogor, N. (2011) The Infobiotics Workbench: An integrated in silico modelling platform for Systems and Synthetic Biology. *Bioinformatics* 27, 3323–3324.
 18. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F. J., Krasnogor, N., and Gheorghe, M. *Applications of Membrane Computing in Systems and Synthetic Biology*; Emergence, Complexity and Computation; Springer International Publishing, 2014; Vol. 7; pp 1–41.
 19. Sanassy, D., Widera, P., and Krasnogor, N. (2015) Meta-Stochastic Simulation of Biochemical Models for Systems and Synthetic Biology. *ACS Synthetic Biology* 4, 39–47.
 20. Jansen, D. N., Katoen, J.-P., Oldenkamp, M., Stoelinga, M., and Zapreev, I. How Fast and Fat is Your Probabilistic Model Checker? An Experimental Performance Comparison. Proceedings of HVC’07. Berlin, Heidelberg, 2008; pp 69–85.
 21. Boyer, B., Corre, K., Legay, A., and Sedwards, S. *Proc. of 10th International Conference QEST*; LNCS; Springer, 2013; Vol. 8054; pp 160–164.

22. Donaldson, R., and Gilbert, D. *A Monte Carlo Model Checker for Probabilistic LTL with Numerical Constraints*; 2008.
23. Zuliani, P. (2014) Statistical Model Checking for Biological Applications. *International Journal on Software Tools for Technology Transfer* 17, 527–536.
24. Kwiatkowska, M., Norman, G., and Parker, D. *Computer performance evaluation: modelling techniques and tools*; Springer Berlin Heidelberg, 2002; pp 200–204.
25. Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. *Tools and Algorithms for the Construction and Analysis of Systems*; Springer Berlin Heidelberg, 2006; pp 441–444.
26. Probabilistic and Symbolic Model Checker (PRISM). <http://www.prismmodelchecker.org/>, n.d.; [Online; accessed 08/01/17].
27. Kwiatkowska, M., Norman, G., and Parker, D. Stochastic Model Checking. Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation. Berlin, Heidelberg, 2007; pp 220–270.
28. Younes, H. L. S. Ymer: A Statistical Model Checker. Proceedings of the 17th International Conference on Computer Aided Verification. Berlin, Heidelberg, 2005; pp 429–433.
29. Zapreev, I. S., and Jansen, C. Markov reward model checker manual. version Version 1.5, January 12, 2011.
30. Aziz, A., Sanwal, K., Singhal, V., and Brayton, R. (2000) Model-checking Continuous-time Markov Chains. *ACM Trans. Comput. Logic* 1, 162–170.
31. Baier, C., Haverkort, B., Hermanns, H., and Katoen, J. P. (2003) Model Checking Algorithms for Continuous-time Markov Chains. *IEEE Transactions on Software Engineering* 29, 524–541.
32. Baier, C., and Katoen, J.-P. *Principles of Model Checking*; The MIT Press, 2008.

33. Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. Patterns in Property Specifications for Finite-state Verification. Proceedings of ICSE '99. New York, NY, USA, 1999; pp 411–420.
34. Grunske, L. Specification Patterns for Probabilistic Quality Properties. Proceedings of the 30th International Conference on Software Engineering. NY, USA, 2008; pp 31–40.
35. Konur, S. (2014) Towards Light-Weight Probabilistic Model Checking. *J. Applied Mathematics 2014*, 814159:1–814159:15.
36. Gheorghe, M., Konur, S., Ipate, F., Mierla, L., Bakir, M. E., and Stannett, M. *Membrane Computing*; Lecture Notes in Computer Science; Springer, 2015; pp 153–170.
37. Konur, S., and Gheorghe, M. (2015) A Property-Driven Methodology for Formal Analysis of Synthetic Biology Systems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics 12*, 360–371.
38. Monteiro, P. T., Ropers, D., Mateescu, R., Freitas, A. T., and de Jong, H. (2008) Temporal Logic Patterns for Querying Dynamic Models of Cellular Interaction Networks. *Bioinformatics 24*, i227–i233.
39. The European Bioinformatics Institute. <http://www.ebi.ac.uk/>, n.d.; [Online; accessed 25/09/16].
40. Newman, M. *Networks: An Introduction*; Oxford University Press, Inc.: New York, NY, USA, 2010.
41. Pavlopoulos, G. A., Secrier, M., Moschopoulos, C. N., Soldatos, T. G., Kossida, S., Aerts, J., Schneider, R., and Bagos, P. G. (2011) Using graph theory to analyze biological networks. *BioData Mining 4*, 10.
42. Ma, H. W., and Zeng, A. P. In *Computational Systems Biology (Second Edition)*; Kriete, A., and Eils, R., Eds.; Academic Press: Oxford, 2014; pp 113 – 131.

43. Wasserman, S., Faust, K., and Iacobucci, D. *Social Network Analysis : Methods and Applications (Structural Analysis in the Social Sciences)*; Cambridge University Press, 1994; p 106.
44. Geurts, P., Ernst, D., and Wehenkel, L. (2006) Extremely Randomized Trees. *Machine Learning* 63, 3–42.
45. Breiman, L., and Cutler, A. Random Forests. http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#giniimp, n.d.; [Online; accessed 28/9/16].
46. DecisionTreeClassifier. n.d.; <http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>, [Online; accessed 28/9/16].
47. Louppe, G., Wehenkel, L., Sutter, A., and Geurts, P. *Advances in Neural Information Processing Systems 26*; Curran Associates, Inc., 2013; pp 431–439.
48. Alpaydin, E. *Introduction to Machine Learning*, 2nd ed.; The MIT Press, 2010.
49. Dummy Estimators. n.d.; http://scikit-learn.org/stable/modules/model_evaluation.html#dummy-estimators, [Online; accessed 01/11/16].
50. Chang, C.-C., and Lin, C.-J. (2011) LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 27:1–27:27.
51. Yu, H.-F., Huang, F.-L., and Lin, C.-J. (2011) Dual Coordinate Descent Methods for Logistic Regression and Maximum Entropy Models. *Mach. Learn.* 85, 41–75.
52. Mucherino, A., Papajorgji, P. J., and Pardalos, P. M. *Data Mining in Agriculture*; Springer: New York, 2009; pp 83–106.
53. Breiman, L. (2001) Random Forests. *Machine Learning* 45, 5–32.
54. Pedregosa, F., Varoquaux, G., Gramfort, A., and et. al., (2011) Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 2825–2830.

55. Scikit-learn. n.d.; <http://scikit-learn.org/stable/>, [Online; accessed 01/11/16].
56. Friedman, M. (1940) A Comparison of Alternative Tests of Significance for the Problem of m Rankings. *Ann. Math. Statist.* 11, 86–92.
57. friedmanchisquare. n.d.; <http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.stats.friedmanchisquare.html>, [Online; accessed 01/11/16].

Part III

Concluding Remarks

Chapter 8

Achievements, Limitations and Future Research

8.1 Achievements

At the beginning of this thesis, we set out our main aim: to simplify the model checking process for researchers in biological systems modelling.

Our *initial objective* was to investigate the use of exact model checking for verifying biological models; a key part of this approach was the development of a software component that can map a high-level modelling language to an exact model checker language (see Section 4.1.1.1). It became apparent, however, that exact model checking is unable to cope with the demands of larger models.

We accordingly shifted focus to look at statistical model checkers, since these are well known to be more capable of dealing with large models. However, even here there were shortcomings since different tools had different performance characteristics and it was not possible to know with any certainty which SMC would be best for each combination of model and query. This informed our *second objective*: to develop a system which can predict, for the verification of any given model and property combination, which SMC is the fastest choice.

We believe we have achieved our aim and objectives, albeit with caveats (discussed below).

8.1.1 Objective 1: Summary of Progress

In the initial stage of this research, we investigated exact model checking for analysis of biological models. We noted that kPWorkbench facilitates modelling and analysis by providing a high-level language for modelling (kP-Lingua), and enables analysis with both simulation and model checking options (see Section 4.1.1). The original model checking component supports verification using the Spin model checker, by accommodating a translator from kP-Lingua specifications to the Spin language, Promela. However, two insufficiencies arise from this reliance on Spin. First, as Spin explicitly builds the entire state space and searches it in an exhaustive brute force manner, it can only verify very small biological models. This means the Spin translator component of kPWorkbench is forced to add an upper bound on the generation of computation steps so as to limit the size of the associated state space. This is a fundamental problem, because this restriction means we can no longer have the full confidence we would normally expect from using the model checking technique. Second, Spin supports only LTL for property specification.

To address these Spin-based shortcomings, we decided to employ the NUSMV model checker [79]. Because NUSMV is a symbolic model checker, it does not construct individual states explicitly. Instead, it symbolically represents states in the form of tree structures, which allows representation of larger state spaces in a compressed form [64]. Additionally, it supports both LTL and CTL for property specification. We accordingly developed a software component which can translate models specified in kP-Lingua into NUSMV specification language. The main challenges faced and the details of the translation system are explained in Sections 1.3.1 and 4.1.1.1. Our NUSMV translator is now the largest component of kPWorkbench as measured in lines of code, and it has successfully been used to verify several models from various different fields, including both biology and engineering; see, e.g., [9, 51, 53, 54, 74].

We believe that by implementing the NUSMV translator we have achieved our *initial objective*; nonetheless, our translator has some limitations – these are discussed in Sections 4.1.1.1 and 8.2.1. To aid dissemination of our work, we also built and continue to maintain a website for kPWorkbench (www.kpworkbench.org) which internally accommodates the latest version of the NUSMV translator. The website also includes several case studies which are modelled in kP-Lingua and their translated NUSMV models.

8.1.2 Objective 2: Summary of Progress

Using the NUSMV translator allows users to verify somewhat larger models (in comparison with the Spin translator). But like all exact model checkers, NUSMV still faced increasingly severe state space limitations as the analysed models grew larger. We accordingly shifted focus to look at statistical model checkers, since these are well known to be more capable of dealing with large models because they consider only a fraction of execution paths and perform model checking through approximation using statistical (e.g., Monte Carlo) methods [12]. The reduced number of paths considered makes the verification of much larger models possible at lower computational cost while trading only a small amount of uncertainty. Thus, SMC blends simulation and model checking by harnessing the speed of simulation and the comprehensive analytical features of model checking.

The encompassing qualities of statistical model checking have led researchers to implement several SMC tools. While this availability of multiple SMC tools provides more flexibility and control to the users, it is associated with a lack of clarity as to which of the tools is best for verification of any particular model and property query. The task of identifying the best SMC tool for a given situation requires a significant degree of verification experience, because each tool has different requirements, capabilities, and performance characteristics. Thus, identification of the best SMC tool becomes an arduous, error-prone, and time-consuming process. Therefore, it is desirable to have another processing layer which can automatically identify the best SMC tool for a given model and property query.

Before attempting to automatically identify the fastest SMC tool for verification of a

given model and property pattern by utilising machine learning algorithms, we checked whether there is a simpler way of achieving good results. We surveyed five popular SMC tools by comparing their modelling languages, external dependencies, expressibility of specification languages, and performances against verification of 475 biological models and five property patterns (see Chapter 6). Our intention was to see if we could identify the fastest model checker by considering only a very few features, i.e. model size and property pattern, so that users could decide which SMC would be the fastest option without needing to use third-party tools. We believe this to be the first attempt to categorise the performance of SMC tools based on property patterns. The experimental results showed that there are cases where we can easily identify the fastest model checker just by looking at the model size and the property pattern—for example, when the model size is either very small or large. However, there are cases where the fastest tool cannot be determined by examining just these two parameters alone. Detailed performance comparisons are provided in Chapter 6 Table 4, Figure [2-6]. These experimental results showed that to identify the fastest SMC prior to model verification, a more extensive feature-set investigation is required. However, this means it is no longer feasible to manually identify the fastest SMC, or to easily formulate the relationship between the features and the fastest SMC.

When the number of the features is high, machine learning algorithms typically offer a suitable solution—sometimes the only solution (see Section 4.2)—because they can gradually learn how to transform inputs to outputs, just by analysing the data. For fastest SMC prediction, the verification result of each model becomes a sample for the machine learning algorithm to learn from, where each sample consists of the model features (as the inputs) and the fastest model checker (as output). These algorithms usually give better results if more data is provided, so we extended the previous performance benchmarking experiment to include more biological models (675) and property patterns (11) (see Chapter 7). More importantly, however, when the identified features are well correlated with the outputs, we are more likely to get better results. Therefore, identifying relevant features is the crucial factor for getting good prediction results. In our paper [12], we identified 12 new model features for predicting the fastest SMC tool for verification of biological models.

To predict the fastest SMC tool for each situation, we provided the performance results and the identified model features to five machine learning algorithms that were selected for their known suitability as classifiers (see Section 4.2). The experimental details and the results are reported in Section 7. The results showed that Extremely Randomized Trees [50] were the best classifier for six property patterns, and Support Vector Machines [33] were best for the other five. Our resulting system could successfully predict the fastest SMC tool for each property pattern with over 90% accuracy, and using our approach also saves users a significant amount of time—up to 208 hours in the experiments we considered!

Finally, we developed a standalone application, **SMC Predictor**, for automating the prediction of the fastest SMC tools for a given model and property query, and developed a dedicated website for SMC predictors www.smcpredictor.com. Installation details, a tutorial, and all experimental data (e.g. biological models, performance results, model features) are available on the website. The predictor’s source code is fully to the public, at www.github.com/meminbakir/smcp.

This shows, we believe, that we have clearly accomplished our second objective, and hence also (subject to certain caveats discussed in the next section) that we have essentially achieved our overall aim.

8.2 Limitations

8.2.1 NuSMV translator

Our NuSMV translator translates kP systems models written in kP-Lingua to NuSMV modelling language specifications (see section 1.3.1, chapter 5 and [53]). However, some of the rules of kP systems—membrane division and dissolution, link creation and destruction rules [53, 74]—can dynamically change the structure of the model, and finding an appropriate translation from such a complex modelling formalism to the model checkers’ specifications is very challenging. The model checking technique is prone to state explosion problems, which generally forces us to use bounded variables. Additionally, NuSMV has

limited support for non-determinism rule execution and array structures. These constraints restricted us to implement a complete translation from kP systems model to NuSMV specifications. Regarding the structure changing rules, the current implementation of NuSMV translator supports membrane dissolution, but only partially supports membrane division, and it does not support the link creation and destruction rules. The membrane division rule is supported only for kP systems models which should always reach the same final compartment and content set. That is, regardless of the execution order, the systems should always end with the same set of compartments and contents. Even models which fulfil this constraint may require many intermediary variables, but since the final state of the systems is always going to be same, these variables would become an extra burden for the state space. Therefore, computing the final set outside of the model checker state space would reduce the complexity of the problem and could potentially help us to save some time and memory. We used the *C#* programming language to externally execute the membrane division rules and obtain the final compartments and contents set. Only the final set is translated to the NuSMV modelling language for further investigation.

8.2.2 SMC prediction

In our research, we used only curated SBML formatted biological models from the BioModels database [43]. In order to focus on the model structure analysis, we have fixed the stochastic rate constants of the reactions to 1.0, and we set the initial population of each species to 50. We also set the upper bound of species to 100, because generally, the model checkers require us to use only bounded variables.

We used 11 of the most-widely used property patterns in our proof-of-concept methodology experiment settings. Although the selected patterns are carefully selected, and they are the most popular patterns, nonetheless the list is not exhaustive. Additionally, we did not allow the union of patterns, as the different combinations could result in very long and complicated queries that would make our experiments intractable.

8.3 Future research directions

Parallellising graph feature analysis: The current implementation of the SMC predictor uses a single thread for feature extraction. It is possible to accelerate the process further by concurrently building the graphs and calculating their properties. The construction of species and reaction graphs are independent, which can be parallelised. The calculation of some of the features also can be parallelised, but not all of them. For example, the node degrees and graph density features can be calculated in parallel. However, both tasks need to wait for the edge number calculation to be completed first.

SMC prediction as a service: The SMC predictor is implemented as a standalone application. As the next step, it can be converted to a web service. This feature will enable larger model analysis platforms, e.g. kPWorkbench and Infobotics Workbench, to easily integrate the SMC predictor into their ecosystem, with a cost of internet dependency and the relevant issues, such as connection speed.

Different domains: Model checking has successfully been applied to various fields, e.g. computer and software systems. Our methodology can be adapted to different systems. For a different domain, researchers would need to develop/employ a high-level modelling language which will be translated to the candidate model checkers' input language. Using the same high-level modelling language will enable extracting the same model features for all candidate model checkers which should be used as input for machine learning algorithms. For example, we employed SBML, as the high-level modelling language. Then, they should identify the most relevant property patterns for the target domain, or they can use the patterns we assembled. For property patterns, also, a high-level property language which can abstract the target model checkers' property languages should be employed. After performance benchmarking of the tools against different property patterns, they should try to identify the determining features which can affect the performance at most. We advise them to experiment with both graph and non-graph related properties. Finally, they can deliver the identified features to more than one machine learning algorithm for

analysing their predictive power.

This approach allows users to try as many model checkers as they want, however, for different systems they will need to develop/employ a custom high-level modelling language. On the other hand, researchers may consider only a subset of model checkers which supports the same modelling language. For the same reason as using a high-level language, namely having the same modelling language will allow extracting the same model features for different model checker tools. In such a case, they can analyse the topological properties of the tools? modelling language, e.g. the number of variables and rules, for performance prediction. Although this approach can accommodate a smaller set of model checker tools, we believe, it can outline more generalised performance characteristics of the tools.

Appendices

Appendix A

A Model Translated using the NuSMV Translator

The following NUSMV model has been obtained using the NuSMV translator; its corresponding kP-Lingua model is provided in Example 2 on page 31. Space limitations only allow us to include a fragment of the model here. The complete translated model, alongside the kP-Lingua model, is available online at www.github.com/meminbakir/kernelP-store.

```
1 MODULE m1_C1 (p_a, p_b, _sync )
2 VAR
3   _status : {_ACTIVE, _willDISSOLVE, _DISSOLVED};
4   _turn   : {_CH00, _ARB1, _MAX2, _SEQ3, _READY};
5   _count  : 0 .. 3;
6   _connC2 : {to_m2, to_m3};
7   _cho0   : 0 .. 2;
8   _arb1   : 0 .. 3;
9   _rand   : 0 .. 3;
10  _arb1_guard1 : boolean;
11  _arb1_guard2 : boolean;
12  _arb1_guard3 : boolean;
13  _max2   : 0 .. 3;
14  _max2_guard1 : boolean;
15  _max2_guard2 : boolean;
16  _max2_guard3 : boolean;
17  _seq3   : 0 .. 2;
18         a : 0 .. 5;
19         a_cp : 0 .. 5;
20         a_m2 : 0 .. 5;
```

```

21     a_m3 : 0 .. 5;
22     b : 0 .. 3;
23     b_cp : 0 .. 3;
24     c : 0 .. 2;
25     c_cp : 0 .. 2;
26
27 INVAR case
28     ((_turn = _CH00) & (((b > 2) & (b >= 2)) |
29         (b >= 1))) : _cho0 != 0;
30     TRUE : _cho0 = 0;
31 esac;
32 INVAR case
33     (_turn = _ARB1) : case
34         (_count = 1) : case
35             (((b >= 1) |
36                 ((a >= 1) & (b >= 1))) |
37                 (c >= 1)) : _arb1 != 0;
38             TRUE : _arb1 = 0;
39         esac;
40         (_count <= _rand) : case
41             (((_arb1_guard1 & (b >= 1)) |
42                 (_arb1_guard2 & ((a >= 1) & (b >= 1)))) |
43                 (_arb1_guard3 & (c >= 1))) : _arb1 != 0;
44             TRUE : _arb1 = 0;
45         esac;
46         TRUE : _arb1 = 0;
47     esac;
48     TRUE : _arb1 = 0;
49 esac;
50 INVAR case
51     (_turn = _MAX2) : case
52         (_count = 0) : case
53             (((b >= 1) |
54                 ((a >= 1) & (b >= 1))) |
55                 (c >= 1)) : _max2 != 0;
56             TRUE : _max2 = 0;
57         esac;
58         (_count = 1) : case
59             (((_max2_guard1 & (b >= 1)) |
60                 (_max2_guard2 & ((a >= 1) & (b >= 1)))) |
61                 (_max2_guard3 & (c >= 1))) : _max2 != 0;
62             TRUE : _max2 = 0;
63         esac;

```

```

64             TRUE : _max2 = 0;
65     esac;
66     TRUE : _max2 = 0;
67 esac;
68
69 ASSIGN
70 init (_status) := _ACTIVE;
71 init (_turn) := _READY;
72 init (_count) := 0;
73 _cho0 := case
74     (_turn = _CH00) : case
75         ((b > 2) & (b >= 2)) : 1;
76         TRUE : 0;
77     esac union case
78         (b >= 1) : 2;
79         TRUE : 0;
80     esac;
81     TRUE : 0;
82 esac;
83 _arb1 := case
84     (_turn = _ARB1) : case
85         (_count = 1) : case
86             (b >= 1) : 1;
87             TRUE : 0;
88         esac union case
89             ((a >= 1) & (b >= 1)) : 2;
90             TRUE : 0;
91         esac union case
92             (c >= 1) : 3;
93             TRUE : 0;
94         esac;
95         (_count <= _rand) : case
96             (_arb1_guard1 & (b >= 1)) : 1;
97             TRUE : 0;
98         esac union case
99             (_arb1_guard2 & ((a >= 1) & (b >= 1))) : 2;
100            TRUE : 0;
101        esac union case
102            (_arb1_guard3 & (c >= 1)) : 3;
103            TRUE : 0;
104        esac;
105        TRUE : 0;
106    esac;

```

```

107         TRUE : 0;
108     esac;
109     init (_rand) := 0;
110     init (_arb1_guard1) := FALSE;
111     init (_arb1_guard2) := FALSE;
112     init (_arb1_guard3) := FALSE;
113     _max2 := case
114         (_turn = _MAX2) : case
115             (_count = 0) : case
116                 (b >= 1) : 1;
117                 TRUE : 0;
118             esac union case
119                 ((a >= 1) & (b >= 1)) : 2;
120                 TRUE : 0;
121             esac union case
122                 (c >= 1) : 3;
123                 TRUE : 0;
124             esac;
125         (_count = 1) : case
126             (_max2_guard1 & (b >= 1)) : 1;
127             TRUE : 0;
128         esac union case
129             (_max2_guard2 & ((a >= 1) & (b >= 1))) : 2;
130             TRUE : 0;
131         esac union case
132             (_max2_guard3 & (c >= 1)) : 3;
133             TRUE : 0;
134         esac;
135         TRUE : 0;
136     esac;
137     TRUE : 0;
138     esac;
139     init (_max2_guard1) := FALSE;
140     init (_max2_guard2) := FALSE;
141     init (_max2_guard3) := FALSE;
142     _seq3 := case
143         (_turn = _SEQ3) : case
144             ((_count = 1) & (a >= 1)) : 1;
145             ((_count = 2) & ((a = 3) & (c >= 2))) : 2;
146             TRUE : 0;
147         esac;
148         TRUE : 0;
149     esac;

```

```

150 init (a) := p_a;
151 init (a_cp) := 0;
152 init (a_m2) := 0;
153 init (a_m3) := 0;
154 init (b) := p_b;
155 init (b_cp) := 0;
156 init (c) := 0;
157 init (c_cp) := 0;
158
159 next (_status) := case
160     ((_status = _willDISSOLVE) & (_sync = _BUSY)) : _willDISSOLVE;
161     ((_status = _willDISSOLVE) & (_sync = _EXCH)) : _DISSOLVED;
162     ((_seq3 = 2) & ((a = 3) & (c >= 2))) : _willDISSOLVE;
163     TRUE : _status;
164 esac;
165
166 next (_turn) := case
167     ((_status = _ACTIVE) & (_turn = _READY)) : case
168         (_sync = _BUSY) : _READY;
169         (_sync = _EXCH) : _CH00;
170         TRUE : _turn;
171     esac;
172     (_turn = _CH00) : _ARB1;
173     (_turn = _ARB1) : case
174         ((_count < _rand) & (_arb1 != 0)) : _ARB1;
175         TRUE : _MAX2;
176     esac;
177     (_turn = _MAX2) : case
178         (_max2 != 0) : _MAX2;
179         TRUE : _SEQ3;
180     esac;
181     (_turn = _SEQ3) : case
182         (_seq3 != 0) : _SEQ3;
183         TRUE : _READY;
184     esac;
185     ((_status = _DISSOLVED)) : _READY;
186     TRUE : _turn;
187 esac;
188 ... //Variables, rules and modules are removed for brevity
189 next (_max2_guard1) := case
190     ((_turn = _MAX2) & ((_max2 != 0) & (_count = 0))) : TRUE;
191     (((_status = _ACTIVE) | ((_status = _willDISSOLVE))) & (next(_turn) = _MAX2)) & (
        _max2 != 0) : _max2_guard1;

```

```

192         TRUE : FALSE;
193     esac;
194     ... //Variables, rules and modules are removed for brevity
195     next (a) := case
196         (((_turn = _ARB1) & (_arb1 = 2)) & (((a - 1 >= 0) & (a - 1 <= 5)) & ((a >= 1) & (b
            >= 1)))) : a - 1;
197         (((_turn = _MAX2) & (_max2 = 2)) & (((a - 1 >= 0) & (a - 1 <= 5)) & ((a >= 1) & (b
            >= 1)))) : a - 1;
198         (((_turn = _SEQ3) & (_seq3 = 1)) & (((a - 1 >= 0) & (a - 1 <= 5)) & (a >= 1))) : a
            - 1;
199         (((_status = _ACTIVE) & (_sync = _EXCH)) & ((a + a_cp >= 0) & (a + a_cp <= 5))) :
            a + a_cp;
200         (((_status = _willDISSOLVE) & (_sync = _EXCH)) : 0;
201         ((_status = _DISSOLVED)) : 0;
202         TRUE : a;
203     esac;
204     next (a_cp) := case
205         (((_turn = _ARB1) & (_arb1 = 3)) & (((a_cp + 1 >= 0) & (a_cp + 1 <= 5)) & (c >= 1)
            )) : a_cp + 1;
206         (((_turn = _MAX2) & (_max2 = 3)) & (((a_cp + 1 >= 0) & (a_cp + 1 <= 5)) & (c >= 1)
            )) : a_cp + 1;
207         ((_status = _ACTIVE) & (_sync = _EXCH)) : 0;
208         (((_status = _willDISSOLVE) & (_sync = _EXCH)) : 0;
209         ((_status = _DISSOLVED)) : 0;
210         TRUE : a_cp;
211     esac;
212     ... //Variables, rules and modules are removed for brevity
213     ----- MAIN -----
214     MODULE main
215     VAR
216     _sync : {_BUSY, _EXCH};
217     pInS : boolean;
218         m1 : m1_C1(2,1,_sync);
219         m2 : m2_C2(1,_sync);
220         m3 : m3_C2(5,_sync);
221     ASSIGN
222     init (_sync) := _EXCH;
223     init (pInS) := TRUE;
224     next (_sync) := case
225         (((next(m1._turn) = _READY))
226         & (next(m2._turn) = _READY))
227         & (next(m3._turn) = _READY)) : _EXCH;
228         TRUE : _BUSY;

```

```
229 esac;
230 next (pInS) := case
231     (_sync = _EXCH) : TRUE;
232     TRUE : FALSE;
233 esac;
234 ...// The rest of the model is truncated.
```

Listing A.1: Code fragments of NUSMV model of Example 2) which is obtained by using NUSMV translator.

Appendix B

Instantiation of the property patterns

The following specifications are used for the instantiation of the property patterns. Please note that for the fastest SMC prediction study the actual instantiation is not very important, as we provide the same property instance to all SMC tools and we evaluate their performance. Therefore, they are tested under the same conditions, as long as the same property is provided to the SMC tools, it is not important whether a property result is ‘true’ or ‘false’. Each of the following queries is expressed in the Pattern Query Language (please see <http://www.smcpredictor.com/pqGrammar.html>). The ‘firstSpecies’ and ‘lastSpecies’ are placeholders represents the first and the last species that appear in the biological models.

```
1  with probability >=1 eventually lastSpecies >=50
2  with probability >=1 always lastSpecies >=50
3  with probability >=1 next lastSpecies >=50
4  with probability >=1 never lastSpecies >50
5  with probability >=1 infinitely-often lastSpecies >=50
6  with probability >=1 steady-state lastSpecies >=50
7  with probability >=1 firstSpecies >=50 until lastSpecies >50
8  with probability >=1 firstSpecies >=50 weak-until lastSpecies >50
9  with probability >=1 lastSpecies >=50 release firstSpecies >=50
10 with probability >=1 lastSpecies >=50 follows firstSpecies >=50
```

```
11 with probability >=1 firstSpecies >=50 precedes lastSpecies >=50
```

Listing B.1: Instantiation of the property patterns

Appendix C

Contribution Statements

The following statements approve the consent of the co-authors of the papers embedded in this thesis. The email correspondences from all co-authors attesting to my contribution to the joint publications have been sighted by my supervisor Dr Mike Stannett.

Contribution Statement

Title	Extended simulation and verification platform for kernel P systems.	
Publication Status	<input checked="" type="checkbox"/> Published	<input type="checkbox"/> Accepted for Publication
	<input type="checkbox"/> Submitted for Publication	<input type="checkbox"/> Unpublished and Unsubmitted work written in a manuscript style
Publication Details	Mehmet E. Bakir, Florentin Ipate, Savas Konur, Laurentiu Mierla, and Ionut Niculescu. Extended simulation and verification platform for kernel P systems. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosik, and Claudio Zandron, editors, <i>Membrane Computing: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers</i> , pages 158–178. Springer International Publishing, Cham, 2014.	
Student Contribution to the Paper	Developed NuSMV translator; performed verification with NuSMV model checker; performed analysis with KPWorkbench simulator; contributed to manuscript writing.	
Student Author's name	Mehmet Emin BAKIR	Signature: 

Co-Author Statement

I hereby declare that I am aware that the work in the paper/manuscript entitled: “**Extended simulation and verification platform for kernel P systems.**” of which I am a co-author, will form part of the PhD dissertation by PhD student: **Mehmet Emin BAKIR** who made a major contribution to the work stated above.

Co-Author Name	Date
Florentin Ipate	20.06.2017
Savas Konur	20.06.2017
Laurentiu Mierla	20.06.2017
Ionut Niculescu	20.06.2017

Supervisor Confirmation

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Name	Dr Mike Stannett	Signature: 
-------------	------------------	--

Contribution Statement

Title	High performance simulations of kernel P systems.	
Publication Status	<input checked="" type="checkbox"/> Published	<input type="checkbox"/> Accepted for Publication
	<input type="checkbox"/> Submitted for Publication	<input type="checkbox"/> Unpublished and Unsubmitted work written in a manuscript style
Publication Details	M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate, "High performance simulations of kernel P systems," in <i>2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICSS)</i> , pp. 409–412, 2014.	
Student Contribution to the Paper	Performed experiments with KPWorkbench simulator, and benchmarked its performance; contributed to manuscript writing.	
Student Author's name	Mehmet Emin BAKIR	Signature: 

Co-Author Statement

I hereby declare that I am aware that the work in the paper/manuscript entitled: "**High performance simulations of kernel P systems.**" of which I am a co-author, will form part of the PhD dissertation by PhD student: **Mehmet Emin BAKIR** who made a major contribution to the work stated above.

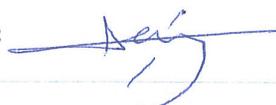
Co-Author Name	Date
Savas Konur	20.06.2017
Marian Gheorghe	20.06.2017
Ionut Niculescu	20.06.2017
Florentin Ipate	20.06.2017

Supervisor Confirmation

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Name	Dr Mike Stannett	Signature: 
-------------	------------------	--

Contribution Statement

Title	Comparative analysis of statistical model checking tools.	
Publication Status	<input checked="" type="checkbox"/> Published <input type="checkbox"/> Accepted for Publication	<input type="checkbox"/> Submitted for Publication <input type="checkbox"/> Unpublished and Unsubmitted work written in a manuscript style
Publication Details	M. E. Bakir, M. Gheorghe, S. Konur, and M. Stannett, "Comparative analysis of statistical model checking tools," in <i>Membrane Computing: 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers</i> (A. Leporati, G. Rozenberg, A. Salomaa, and C. Zandron, eds.), pp. 119–135, Cham: Springer International Publishing, 2017.	
Student Contribution to the Paper	Conducted literature review of SMC tools; benchmarked the performance the tools; interpreted findings; contributed to manuscript writing.	
Student Author's name	Mehmet Emin BAKIR	Signature: 

Co-Author Statement

I hereby declare that I am aware that the work in the paper/manuscript entitled: "**Comparative analysis of statistical model checking tools.**" of which I am a co-author, will form part of the PhD dissertation by PhD student: **Mehmet Emin BAKIR** who made a major contribution to the work stated above.

Co-Author Name	Date
Marian Gheorghe	20.06.2017
Savas Konur	20.06.2017
Mike Stannett	20.06.2017

Supervisor Confirmation

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Name	Dr Mike Stannett	Signature: 
-------------	------------------	--

Contribution Statement

Title	Performance benchmarking and automatic selection of verification tools for efficient analysis of biological models.	
Publication Status	<input type="checkbox"/> Published	<input type="checkbox"/> Accepted for Publication
	<input type="checkbox"/> Submitted for Publication	<input checked="" type="checkbox"/> Unpublished and Unsubmitted work written in a manuscript style
Publication Details	Mehmet Emin Bakir, Savas Konur, Marian Gheorghe, Natalio Krasnogor, and Mike Stannett. "Performance benchmarking and automatic selection of verification tools for efficient analysis of biological models.", unpublished, 2017.	
Student Contribution to the Paper	Conducted experiments; interpreted findings; contributed to manuscript writing.	
Student Author's name	Mehmet Emin BAKIR	Signature: 

Co-Author Statement

I hereby declare that I am aware that the work in the paper/manuscript entitled: "**Performance benchmarking and automatic selection of verification tools for efficient analysis of biological models.**" of which I am a co-author, will form part of the PhD dissertation by PhD student: **Mehmet Emin BAKIR** who made a major contribution to the work stated above.

Co-Author Name	Date
Savas Konur	20.06.2017
Marian Gheorghe	20.06.2017
Natalio Krasnogor	20.06.2017
Mike Stannett	20.06.2017

Supervisor Confirmation

I have sighted email or other correspondence from all co-authors confirming their certifying authorship.

Name	Dr Mike Stannett	Signature: 
-------------	------------------	--

References

- [1] Gul Agha and Karl Palmskog. A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.*, 28(1):6:1–6:39, January 2018.
- [2] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.
- [3] Marco Antonioti, Alberto Policriti, Nadia Ugel, and Bud Mishra. Model building and model checking for biochemical processes. *Cell Biochemistry and Biophysics*, 38(3):271–286, 2003.
- [4] Kathryn Atwell, Sara-Jane Dunn, James M. Osborne, Hillel Kugler, and E. Jane Albert Hubbard. How computational models contribute to our understanding of the germ line. *Molecular Reproduction and Development*, 83(11):944–957, 2016.
- [5] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic*, 1(1):162–170, July 2000.
- [6] C. Baier, B. Haverkort, H. Hermanns, and J. P. Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, June 2003.
- [7] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

- [8] M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High performance simulations of kernel P systems. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICSS)*, pages 409–412, 2014.
- [9] Mehmet E. Bakir, Florentin Ipate, Savas Konur, Laurentiu Mierla, and Ionut Niculescu. Extended simulation and verification platform for kernel P systems. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *Membrane Computing: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers*, pages 158–178. Springer International Publishing, Cham, 2014.
- [10] Mehmet E. Bakir and Mike Stannett. Selection criteria for statistical model checking. In M Gheorghe and S Konur, editors, *Proceedings of the Workshop on Membrane Computing WMC 2016, Manchester (UK), 11-15 July 2016*, pages 55–57, 2016. Available as: Technical Report UB-20160819-1, University of Bradford.
- [11] Mehmet Emin Bakir, Marian Gheorghe, Savas Konur, and Mike Stannett. Comparative analysis of statistical model checking tools. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing: 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers*, pages 119–135. Springer International Publishing, Cham, 2017.
- [12] Mehmet Emin Bakir, Savas Konur, Marian Gheorghe, Natalio Krasnogor, and Mike Stannett. Performance benchmarking and automatic selection of verification tools for efficient analysis of biological models. Unpublished, 2017.
- [13] Mehmet Emin Bakir, Savas Konur, Marian Gheorghe, Natalio Krasnogor, and Mike Stannett. Automatic selection of verification tools for efficient analysis of biochemical models. *Bioinformatics*, 2018. Available online: <https://academic.oup.com/bioinformatics/advance-article/doi/10.1093/bioinformatics/bty282/4983061>.

- [14] Paweł Banasik, Anna Gambin, Sławomir Lasota, Michał Lula, and Mikołaj Rybiński. Tav4sb: integrating tools for analysis of kinetic models of biological systems. *BMC Systems Biology*, 6(25), 2012.
- [15] Ezio Bartocci and Pietro Lió. Computational modeling, formal analysis, and tools for systems biology. *PLoS Comput Biol*, 12(1):e1004591, 2016.
- [16] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, 2006.
- [17] Jacob Beal, Andrew Phillips, Douglas Densmore, and Yizhi Cai. High-level programming languages for biomolecular systems. In *Design and analysis of biomolecular circuits*, pages 225–252. Springer, 2011.
- [18] Mordechai Ben-Ari. *Mathematical logic for computer scientists*. Springer, London, 1 edition, 2009.
- [19] Francesco Bernardini, Marian Gheorghe, Francisco José Romero-Campero, and Neil Walkinshaw. A hybrid approach to modeling biological 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 138–159. Springer Berlin Heidelberg, 2007.
- [20] Daniela Besozzi, Paolo Cazzaniga, Giancarlo Mauri, and Dario Pescini. Biosimware: A software for the modeling, simulation and analysis of biological systems. In Marian Gheorghe, Thomas Hinze, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 119–143. Springer Berlin Heidelberg, 2011.
- [21] Lesia Bilitchenko, Adam Liu, Sherine Cheung, Emma Weeding, Bing Xia, Mariana Leguia, J. Christopher Anderson, and Douglas Densmore. Eugene – a domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE*, 6(4):e18882, 04 2011.
- [22] Biobricks Foundation. <http://biobricks.org/>, n.d. [Online; accessed 7/02/15].

- [23] The Biochemical Abstract Machine (Biocham). <http://lifeware.inria.fr/biocham/DOC/manual.html>, n.d. [Online; accessed 23/01/15].
- [24] J. Blakes, J. Twycross, Francisco José Romero-Campero, and N. Krasnogor. The Infobiotics Workbench: An integrated in silico modelling platform for systems and synthetic biology. *Bioinformatics*, 27(23):3323–3324, December 2011.
- [25] Jonathan Blakes, Jamie Twycross, Savas Konur, Francisco José Romero-Campero, Natalio Krasnogor, and Marian Gheorghe. Infobiotics Workbench: A P systems based tool for systems and synthetic biology. In Pierluigi Frisco, Marian Gheorghe, and Mario J. Pérez-Jiménez, editors, *Applications of Membrane Computing in Systems and Synthetic Biology*, volume 7 of *Emergence, Complexity and Computation*, pages 1–41. Springer International Publishing, 2014.
- [26] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [27] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.
- [28] Luboš Brim, Milan Češka, and David Šafránek. Model checking of biological systems. In Marco Bernardo, Erik de Vink, Alessandra Di Pierro, and Herbert Wiklicky, editors, *Formal Methods for Dynamical Systems: 13th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2013, Bertinoro, Italy, June 17-22, 2013. Advanced Lectures*, pages 63–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [29] G. Wayne Brodland. How computational models can help unlock biological systems. *Seminars in Cell & Developmental Biology*, 47–48:62 – 73, 2015.
- [30] Yizhi Cai, Brian Hartnett, Claes Gustafsson, and Jean Peccoud. A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. *Bioinformatics*, 23(20):2760–2767, 2007.

- [31] Yizhi Cai, Mandy L. Wilson, and Jean Peccoud. GenoCAD for iGEM: a grammatical approach to the design of standard-compliant constructs. *Nucleic Acids Research*, 38(8):2637–2644, May 2010.
- [32] Miguel Carrillo, Pedro A G3ngora, and David A Rosenblueth. An overview of existing modeling tools making use of model checking in the analysis of biochemical networks. *Frontiers in Plant Science*, 3(155):1–13, 2012.
- [33] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.
- [34] A. Cimatti and M. Roveri. NuSMV 1.0: User manual. Technical report, ITC-IRST, Trento, Italy, December 1998.
- [35] E. M. Clarke, O Grumberg, and D Peled. *Model checking*. MIT Press, 1999.
- [36] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
- [37] Simon Coakley, Marian Gheorghe, Mike Holcombe, Shawn Chin, David Worth, and Chris Greenough. Exploitation of high performance computing in the FLAME agent-based simulation framework. In *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPC-ICSS 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 538–545, 2012.
- [38] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69 – 110, 2004. Computational Systems Biology.
- [39] Pedro Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, October 2012.
- [40] D. Donaldson, R.;Gilbert. A Monte Carlo model checker for Probabilistic LTL with numerical constraints. Technical report, University of Glasgow, Department of Computing Science, 2008.

- [41] Ciprian Dragomir, Florentin Ipate, Savas Konur, Raluca Lefticaru, and Laurentiu Mierla. Model checking kernel P systems. In *Membrane Computing*, volume 8340 of *Lecture Notes in Computer Science*, pages 151–172. Springer Berlin Heidelberg, 2014.
- [42] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering, ICSE '99*, pages 411–420, New York, NY, USA, 1999. ACM.
- [43] The European Bioinformatics Institute. <http://www.ebi.ac.uk/>, n.d. [Online; accessed 25/09/16].
- [44] E A Emerson. Temporal and modal logic. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, Amsterdam, 1990.
- [45] Eugene. <http://www.eugencad.org/>, n.d. [Online; accessed 6/02/15].
- [46] Jasmin Fisher and Thomas A Henzinger. Executable cell biology. *Nat Biotech*, 25(11):1239–1249, 2007.
- [47] Flexible Large-scale Agent Modelling Environment (FLAME). <http://www.flame.ac.uk/>, n.d. [Online; accessed 30/01/15].
- [48] Pierluigi Frisco, Marian Gheorghe, and Mario J. Pérez-Jiménez, editors. *Applications of Membrane Computing in Systems and Synthetic Biology*, volume 7 of *Emergence, Complexity and Computation*. Springer International Publishing, 2014.
- [49] GenoCAD . <http://www.genocad.org/>, n.d. [Online; accessed 6/02/15].
- [50] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.
- [51] Marian Gheorghe, R. Ceterchi, F. Ipate, Savas Konur, and Raluca Lefticaru. Kernel P systems: from modelling to verification and testing. *Theoretical Computer Science*, 2017. Accepted for publication.

- [52] Marian Gheorghe, Florentin Ipate, Ciprian Dragomir, Laurentiu Mierla, Luis Valencia-Cabrera, Manuel Garcia-Quismondo, and Mario J Pérez-Jiménez. Kernel P systems – version 1. *Eleventh Brainstorming Week on Membrane Computing, BWMC 2013*, pages 97–124, 2013.
- [53] Marian Gheorghe, Florentin Ipate, Laurentiu Mierla, and Savas Konur. kPWorkbench: a software framework for kernel P systems. In L.F. Macías-Ramos, G. Păun, A. Riscos-Núñez, and L. Valencia-Cabrera, editors, *Thirteenth Brainstorming Week on Membrane Computing, BWMC 2015*, pages 179–194, Sevilla, Spain, 2015. Fénix Editora.
- [54] Marian Gheorghe, Savas Konur, Florentin Ipate, Laurentiu Mierla, Mehmet E. Bakir, and Mike Stannett. An integrated model checking toolset for kernel P systems. In *Membrane Computing: 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers*, pages 153–170. Springer International Publishing, Cham, 2015.
- [55] Marian Gheorghe, Vincenzo Manca, and Francisco José Romero-Campero. Deterministic and stochastic P systems for modelling cellular processes. *Natural Computing*, 9(2):457–473, 2010.
- [56] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403 – 434, 1976.
- [57] Genetic Network Analyzer (GNA). <http://www.genostar.com/category/products/gna/>, n.d. [Online; accessed 31/01/15].
- [58] Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 31–40, NY, USA, 2008. ACM.
- [59] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003.
- [60] James W. Haefner. *Models of Systems*, pages 3–16. Springer US, Boston, MA, 2005.

- [61] Monika Heiner, David Gilbert, and Robin Donaldson. Petri nets for systems and synthetic biology. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *Proceedings of the Formal Methods for the Design of Computer, Communication, and Software Systems 8th International Conference on Formal Methods for Computational Systems Biology (SFM 2008)*, volume 5016 of *LNCS*, pages 215–264, Berlin, Heidelberg, 2008. Springer-Verlag.
- [62] Thomas Héruault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 73–84, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [63] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le Novère, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.
- [64] M Huth and M Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.
- [65] Oscar H Ibarra and Gheorghe Păun. Membrane computing: A general view. *Annals of the European Academy of Sciences*, pages 83–101, 2006.
- [66] Infobiotics. <http://www.infobiotics.org/>, n.d. [Online; accessed 08/01/15].
- [67] Sumit K. Jha, Edmund M. Clarke, Christopher J. Langmead, Axel Legay, André Platzer, and Paolo Zuliani. A bayesian approach to model checking biological systems. In Pierpaolo Degano and Roberto Gorrieri, editors, *Computational Methods in Systems Biology*, pages 218–234, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

- [68] Grace P Julia and G Jeyakumar. Learning the modeling of endocytic pathway of transferrin-bound iron to cells using P systems. *International Journal of Systems Biology*, 3(1):30–33, 2012.
- [69] Savas Konur, Michael Fisher, and Sven Schewe. Combined model checking for temporal, probabilistic, and real-time logics. *Theoretical Computer Science*, 503:61–88, September 2013.
- [70] Savas Konur, Marian Gheorghe, Ciprian Dragomir, Florentin Ipate, and Natalio Krasnogor. Conventional verification for unconventional computing: A genetic XOR gate example. *Fundam. Inf.*, 134(1-2):97–110, January 2014.
- [71] Savas Konur, Marian Gheorghe, Ciprian Dragomir, Laurentiu Mierla, Florentin Ipate, and Natalio Krasnogor. Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. *ACS Synthetic Biology*, 4(1):83–92, 2015.
- [72] kPWorkbench. <http://kpworkbench.org/>, n.d. [Online; accessed 25/09/16].
- [73] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation*, SFM’07, pages 220–270, Berlin, Heidelberg, 2007. Springer-Verlag.
- [74] Raluca Lefticaru, Mehmet Emin Bakir, Savas Konur, Mike Stannett, and Florentin Ipate. Modelling and validating an engineering application in kernel P systems. In Marian Gheorghe, Savas Konur, and Raluca Lefticaru, editors, *Proceedings of the 18th International Conference on Membrane Computing (CMC18)*, pages 205–217. University of Bradford, July 2017.
- [75] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *Proceedings of Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4*, pages 122–135, Berlin, Heidelberg, 2010. Springer.
- [76] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding variable importances in forests of randomized trees. In *Advances in Neural Information Processing Systems 26*, pages 431–439. Curran Associates, Inc., 2013.

- [77] Robin Milner. *Communicating and mobile systems: The Pi-calculus*. Cambridge University Press, New York, NY, USA, 1999.
- [78] Antonio Mucherino, Petraq J. Papajorgji, and Panos M. Pardalos. k-nearest neighbor classification. In *Data Mining in Agriculture*, pages 83–106. Springer New York, New York, NY, 2009.
- [79] NuSMV. <http://nusmv.fbk.eu/>, n.d. [Online; accessed 18/07/14].
- [80] nuXmv. <https://nuxmv.fbk.eu/>, n.d. [Online; accessed 08/01/15].
- [81] National Research Council (US) Committee on Frontiers at the Interface of Computing and Biology. Computational modeling and simulation as enablers for biological discovery. In JC Wooley and HS Lin, editors, *Catalyzing inquiry at the interface of computing and biology*, pages 117–204. National Academies Press (US), Washington, DC, 2005.
- [82] Pattern Query Language (PQL). <http://www.smcpredictor.com/pqGrammar.html>, n.d. [Online; accessed 11/03/2018].
- [83] Gheorghe Păun. *Membrane Systems with Symbol—Objects*, pages 51–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
- [84] Gheorghe Păun. Introduction to membrane computing. In Gabriel Ciobanu, Gheorghe Păun, and Mario J. Pérez-Jiménez, editors, *Applications of Membrane Computing*, Natural Computing Series, pages 1–42. Springer Berlin Heidelberg, 2006.
- [85] Gheorghe Păun and Mario J. Pérez-Jiménez. Membrane computing: Brief introduction, recent results and applications. *Biosystems*, 85(1):11 – 22, 2006. Dedicated to the Memory of Ray Paton.
- [86] Michael Pedersen and Andrew Phillips. Towards programming languages for genetic engineering of living cells. *Journal of The Royal Society Interface*, 6(Suppl 4):S437–S450, 2009.
- [87] Michael Pedersen and Andrew Phillips. Visual GEC Manual. <http://research.microsoft.com/en-us/projects/gec/manual.pdf>, n.d. [Online; accessed 6/02/15].

- [88] Michael Pedersen and Gordon D. Plotkin. A language for biochemical systems: Design and formal specification. In Corrado Priami, Rainer Breitling, David Gilbert, Monika Heiner, and Adelinde M. Uhrmacher, editors, *Transactions on Computational Systems Biology XII*, volume 5945 of *Lecture Notes in Computer Science*, pages 77–145. Springer Berlin Heidelberg, 2010.
- [89] Francisco Pereira, Tom Mitchell, and Matthew Botvinick. Machine learning classifiers and fMRI: A tutorial overview. *NeuroImage*, 45(1, Supplement 1):S199–S209, 2009. Mathematics in Brain Imaging.
- [90] Ignacio Pérez-Hurtado, Luis Valencia-Cabrera, Mario J. Pérez-Jiménez, M. Angels Colomer, and Agustín Riscos-Núñez. Mecosim: A general purpose software tool for simulating biological phenomena by means of P systems. *IEEE Fifth International Conference on Bio-inspired Computing: Theories and Applications (BIC-TA 2010)*, I:637–643, 2010.
- [91] Andrew Phillips. Simulating biological systems in the stochastic pi-calculus. Technical report, Microsoft Research, 2004.
- [92] Platform Independent Petri net Editor 2 (PIPE2). <http://pipe2.sourceforge.net/>, n.d. [Online; accessed 08/01/15].
- [93] C. Priami. Stochastic π -calculus. *The Computer Journal*, 38(7):578–589, 1995.
- [94] Corrado Priami. Algorithmic systems biology. *Commun. ACM*, 52(5):80–88, May 2009.
- [95] Corrado Priami. Algorithmic systems biology — computer science propels systems biology. In Grzegorz Rozenberg, Thomas Back, and Joost N. Kok, editors, *Handbook of Natural Computing*, pages 1835–1862. Springer Berlin Heidelberg, 2012.
- [96] Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, October 2001.
- [97] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-validation. In LING LIU and M. TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 532–538. Springer US, Boston, MA, 2009.

- [98] Wolfgang Reisig. The basic concepts. In *Understanding Petri Nets*, pages 13–24. Springer Berlin Heidelberg, 2013.
- [99] Francisco José Romero-Campero, Marian Gheorghe, Gabriel Ciobanu, John M. Auld, and Mario J. Pérez-Jiménez. Cellular modelling using P systems and process algebra. *Progress in Natural Science*, 17:375–383, 2007.
- [100] Probabilistic and Symbolic Model Checker (PRISM). <http://www.prismmodelchecker.org/>, n.d. [Online; accessed 08/01/15].
- [101] Mikołaj Rybiński, Michał Lula, , Sławomir Lasota, and Anna Gambin. Tav4sb: grid environment for analysis of kinetic models of biological systems. In *Proc. ISBRA'11 Short Abstract*, pages 92–95, 2011.
- [102] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959.
- [103] Daven Sanassy, Harold Fellermann, Natalio Krasnogor, Savas Konur, Laurentiu Mierlă, Marian Gheorghe, Christophe Ladroue, and Sara Kalvala. Modelling and stochastic simulation of synthetic biological Boolean gates. In *16th IEEE International Conference on High Performance Computing and Communications, HPCCC '14*, pages 404–408, Paris, France, 2014. IEEE.
- [104] Daven Sanassy, Paweł Widera, and Natalio Krasnogor. Meta-stochastic simulation of biochemical models for systems and synthetic biology. *ACS Synthetic Biology*, 4(1):39–47, 2015.
- [105] Davide Sangiorgi and David Walker. *Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.
- [106] Herbert M. Sauro and Boris N. Kholodenko. Quantitative analysis of signaling networks. *Progress in Biophysics and Molecular Biology*, 86(1):5 – 43, 2004. New approaches to modelling and analysis of biochemical reactions, pathways and networks.

- [107] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 202–215, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [108] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 266–280, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [109] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, New York, NY, USA, 2014.
- [110] Lucian P. Smith, Frank T. Bergmann, Deepak Chandran, and Herbert M. Sauro. Antimony: A modular model definition language. *Bioinformatics*, 2009.
- [111] Stephen Smith and Ramon Grima. Breakdown of the reaction-diffusion master equation with nonelementary rates. *Phys. Rev. E*, 93:052135, May 2016.
- [112] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.
- [113] Spin - Formal Verification. <http://spinroot.com/spin/whatispin.html>, n.d. [Online; accessed 18/07/14].
- [114] P. Umesh, F. Naveen, Chanchala Uma Maheswara Rao, and Achuthsankar S. Nair. Programming languages for synthetic biology. *Systems and Synthetic Biology*, 4(4):265–269, 2010.
- [115] A. Wald. Sequential tests of statistical hypotheses. *Ann. Math. Statist.*, 16(2):117–186, 06 1945.
- [116] Mandy L. Wilson, Russell Hertzberg, Laura Adam, and Jean Peccoud. Chapter eight - a step-by-step introduction to rule-based design of synthetic genetic constructs using genocad. In Christopher Voigt, editor, *Synthetic Biology, Part B Computer Aided Design and DNA Assembly*, volume 498 of *Methods in Enzymology*, pages 173 – 188. Academic Press, 2011.

- [117] Verena Wolf, Rushil Goel, Maria Mateescu, and Thomas A. Henzinger. Solving the chemical master equation using sliding windows. *BMC Systems Biology*, 4(1):42, Apr 2010.
- [118] H. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):216–228, 2006.
- [119] H. L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005.
- [120] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Proceedings of Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31*, pages 223–235. Springer, Berlin, Heidelberg, 2002.
- [121] Hsiang-Fu Yu, Fang-Lan Huang, and Chih-Jen Lin. Dual coordinate descent methods for logistic regression and maximum entropy models. *Mach. Learn.*, 85(1-2):41–75, October 2011.