This research was funded by the VU University Amsterdam.

This work was carried out in the ASCI graduate school. ASCI dissertation series number: 259.

Copyright © 2012 by Elżbieta Krepska.

Cover design by Elżbieta Krepska.


Printed and bound by Smart Printing Solutions, Gouda, Netherlands.
TOWARDS BIG BIOLOGY:
HIGH-PERFORMANCE VERIFICATION
OF LARGE CONCURRENT SYSTEMS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op woensdag 5 september 2012 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Elżbieta Łucja Krepska

geboren te Warschau, Polen
promotoren: prof.dr.ir. H.E. Bal
           prof.dr. W.J. Fokkink
copromotor: Dr.-Ing.habil. T. Kielmann
Acknowledgments

It was a pleasure to work with all the wonderful people at the VU University Amsterdam. First and foremost, I would like to thank my advisers: Henri Bal, Wan Fokkink and Thilo Kielmann, who offered me continuous support, excellent guidance, and opened doors to many opportunities. I learned a lot during this time, and acquired invaluable experience in research and industry.

I would also like to express my gratitude to our collaborators in the Centre for Integrative Bioinformatics at the VU, who co-authored a chapter of this thesis and strongly influenced all chapters. I greatly enjoyed the time I spent working with Nicola Bonzanni, and I appreciate the support as well as many hours of wide-ranging discussions with Jaap Heringa and Anton Feenstra. I also thank Nicola and his parents for hosting me in Milano—grazie!

Next, I would like to thank the members of my thesis committee: Jiří Barnat (Masaryk University, Brno, Czech Republic), Dragan Bošnački (Eindhoven University of Technology, Netherlands), Byron Cook (Microsoft Research Cambridge and University College London, UK), Jaap Heringa (VU) and Ana Varbanescu (Delft University of Technology, Netherlands), for their time, and comments that helped to improve the thesis.

During the course of my Ph.D., I had a privilege to work at two institutes outside of the university. I am grateful to Jasmin Fisher and Byron Cook for letting me join their groups at Microsoft Research Cambridge—I learned a lot from them. I also thank Nir Piterman (University of Leicester, UK), Antje Beyer and Samin Ishtiaq for a pleasant work atmosphere and a fruitful collaboration. Next, I am thankful to Jim Dehnert for inviting me to join Google as an intern, and for being a great host. Although my work at Google is not part of this thesis, I learned a lot there, broadened my horizons, and had a great time in California, for which I would like to acknowledge the entire Pregel team: Matt Austern, Aart Bik, Michał Węgiel and Grzegorz Czajkowski.

A great number of other people contributed to the completion this thesis. I thank Stefan Vijzelaar (I reported so many bugs and he never complained!) and Ceriel Jacobs for help with software, Erik de Vink, Dragan Bošnački, and Kees van Reeuwijk for interesting discussions, and Kees Verstoep for being the best admin one can imagine. I am also grateful to the entire HPDC team—especially to my office mates: Sridith, Reza, Maik, Michel and Kees—for making me feel welcome at the VU, and to Ilse and Caroline for being always helpful.

Last but not least, I wholeheartedly thank my friends and family, especially Aleksander, for supporting me all these years, motivating me to achieve my goals, and not letting me work after a day long ended and I lost track of time.
We now have unprecedented means of collecting data at the deepest molecular level of living systems and we have relatively cheap and accessible computer power to store and analyze this information. There is, however, a general sense that understanding all this information has lagged far behind its accumulation.

SYDNEY BRENNER

*Sequences and consequences*, 2010

S. Brenner is a Nobel prize laureate in physiology and medicine, and the proponent of *C. elegans* as a model organism.
# Contents

Acknowledgments v

## Contents

ix

1 General introduction 1

1.1 Big biology ................................................. 1
1.2 Modeling in biology ........................................ 1
1.3 Goals and challenges ....................................... 4
1.4 Contributions ................................................. 4
1.5 Outline of this dissertation ................................. 5
1.6 Collaborations ................................................. 5
1.7 Bibliography ................................................. 5

2 Monte Carlo simulation of multi-cellular development 9

2.1 Introduction .................................................. 9
2.2 Related work ................................................ 11
2.3 Biological interpretation of Petri nets ...................... 11
2.4 C. elegans vulval development .............................. 14
2.5 Modeling multi-cellular development ....................... 16
2.6 Simulation procedure ....................................... 18
2.7 Monte Carlo verification .................................... 20
2.8 Verification results ......................................... 21
2.9 Biological results ........................................... 24
2.10 Summary and conclusions .................................. 27
2.11 Bibliography ................................................. 27

3 Proving stabilization by abstract interpretation 31

3.1 Introduction .................................................. 31
3.2 Related work ................................................ 34
3.3 Example: Skin cells ......................................... 35
3.4 Preliminaries ................................................ 39
3.5 Stabilization algorithm ..................................... 40
3.6 Formal proofs ............................................... 45
3.7 Experimental results ....................................... 46
3.8 Summary and conclusions .................................. 50
3.9 Bibliography ................................................. 50
4 Distributed processing of large graphs 55
4.1 Introduction ........................................... 55
4.2 Related work ........................................ 58
4.3 Basic model and API ................................. 60
4.4 Divide-and-conquer graph algorithms ............ 63
4.5 On-the-fly graph algorithms ....................... 64
4.6 Implementation ..................................... 67
4.7 Evaluation ........................................... 69
4.8 Summary and conclusions .......................... 75
4.9 Bibliography ........................................ 76

5 Distributed search for terminal strongly connected components 81
5.1 Introduction ........................................... 81
5.2 Related work ........................................ 82
5.3 Preliminaries ........................................ 83
5.4 Reachability versus strongly connected components ... 84
5.5 The TSCC\textsubscript{DC} algorithm ................ 86
5.6 Implementation ..................................... 86
5.7 Evaluation .......................................... 88
5.8 Summary and conclusions ......................... 93
5.9 Bibliography ........................................ 93

6 Summary and conclusions 97
6.1 Summary of the thesis .............................. 97
6.2 Summary of contributions .......................... 98
6.3 Conclusions ......................................... 99
6.4 Future outlook ...................................... 99

Publications 101

Samenvatting (Summary in Dutch) 103
1.1 Big biology

Biologists traditionally tend to specialize in narrow areas within their field: one cell, one process, one organism; they often study the selected processes in isolation, abstracting away many interactions for tractability. On the one hand, this methodology has brought numerous breakthroughs, such as vaccinations and in vitro fertilization, the two discoveries honored with a Nobel prize in medicine in 2010 and 2011 [1]. On the other hand, it encourages understanding an organism as a bag of loosely-connected parts, and thus may lead to missing vital system-level insights [2]. Systems biology is a relatively young field that aims to describe the entirety of processes within a living organism, and grasp the emergent properties of such processes combined. This approach necessarily involves dealing with systems that model entities at different scales: molecules, cells, organs, organisms; as well as with systems of large size: thousands of concurrently-executing components.

The holistic approach of systems biology is feasible today, in the era of big computers and big data, more than ever before. Powerful computers have become inexpensive and thus easily available, which coincided with the advances in high-throughput biological technologies, such as genome sequencing [3]. Being able to obtain copious amounts of data is, nevertheless, insufficient to gain a deep understanding of biology [4, 5]; rather, big data makes scientists realize the need for modeling, supported by computer analysis, in order to extract knowledge from the data [4, 6]. The challenge is not anymore to obtain more data; the challenge is to use big data and big computers to do big biology.

1.2 Modeling in biology

Models are the most important tool of systems biology. A model is an unambiguous, i.e. formal, representation of a biological system, at a chosen level of abstraction. In molecular cell biology, the players include: genes, DNA,
proteins, membranes, etc.; models describe relations between the players. A model is *executable* [7] if it also represents dynamics of the system, such as changes in protein levels, or movement of molecules through membranes. In general, there are two kinds of models: *continuous* and *discrete*. In continuous models variables are real numbers; such models are mainly expressed as differential equations. In discrete models, quantities are discrete: variables either explicitly denote the number of molecules [8, 9], or are discretized to several predefined concentration levels [7, 10, 11]. Many methods of expressing discrete executable models are in use, for example Petri nets [12], process algebras [9], Boolean/Qualitative/Regulatory Networks [13, 14], stochastic systems [15] and others—see [16] for a detailed discussion. Choosing a suitable formalism depends on the modeler’s intuition and objectives, as different formalisms are amenable to different analysis techniques.

Why would a biologist want to use formal models? For a start, formal models are an excellent way to store and share knowledge on biological systems, and to reason about such systems. Furthermore, experiments *in silico* (in a computer), compared to the wet lab experiments, are cheaper, faster and require less labor, as well as pose no ethical dilemmas; for example, Bonzanni and I carried out thousands of *in-silico* experiments on the *C. elegans* worm with a particularly lethal genetic makeup (*e.g.* Experiment 5, see Section 2.8), while only few could have been realized in wet labs. Most importantly, we were also able to execute computer experiments that would be impossible to perform in a wet lab, such as selectively removing parts of the system, or altering genes during an experiment; such experiments are called *predictive* and are the holy grail of systems biology. Last but not least, our experience and experience of others [17] show that the very process of building models leads to biological insights, when the researcher is forced to rigorously formulate the studied system.

In order to make model-based predictions about biology, one must be convinced that the models faithfully represent nature; to this end, the models must be *verified*: checked for agreement against the known biological evidence. The process of working with models is displayed in Figure 1.1, and was discussed at length in [16]. The model is a working hypothesis (A) representing the current understanding of the studied system. It has to be corrected and calibrated until it passes a suite of verification tests (B) that check that the *in-silico* experiments accurately reproduce the phenotypes (characteristics) and behavior of the original system, with respect to the known wet lab experiments. Afterwards, the predictive *in-silico* experiments can be performed (C), which allows making hypotheses (D) about the underlying biological processes. The conjectures are subsequently proved or disproved in the wet lab, and the modeling cycle restarts when the model is altered to account for the new facts.

We introduce verification (stage (B) in Figure 1.1) in more detail, as it is the focus of this dissertation. The biological facts to check are typically ‘input-output’ rules, where an input is an initial setup of the system, and an output is the corresponding expected outcome. For instance, in the organ development process studied in Chapter 2, the inputs contain 60 genetic perturbations previously tested in a wet lab; for each genetic perturbation the expected
animal’s phenotype is specified in the form of the locations of the developed organs. Another example is in Chapter 3, in which the inputs are initial concentration levels of proteins in skin cells, and outputs are the reached protein levels, from which it can be determined if a cell is proliferating or not. The aim of verification is to check that the model behaves according to these biological ‘rules’. If verification succeeds, we obtain a degree of confidence that the model faithfully represents the biological process studied. In contrast, failed verification suggests a bug in the model, or a gap in understanding of the underlying biology.

Discrete and continuous models require in general very different verification tools. In this research we target discrete systems; we do not address verification of continuous systems, where—since differential equations are deterministic—it is enough to execute such a model once, i.e. integrate the equations. Determinism in this case is a consequence of the assumption that the studied system is an ‘evenly-mixed soup’—in essence the model describes the system’s average behavior. The same assumption is made by stochastic simulations [15], also not addressed in this dissertation, which simulate chemical reactions in a closed volume. Both differential equations and stochastic simulations have two serious limitations: first, any system with membranes, such as a cell, renders the even-mixing assumption invalid; second, these techniques require precise values for constants and rates that are mostly unavailable [18]. In practice biological systems are unevenly distributed, and expressed in qualitative rather than quantitative terms.

An important technique to rigorously verify discrete biological systems is model checking [19–21], in which temporal properties about a formal system are checked by systematically examining the state space of the system. A temporal property is expressed in terms of time—in this dissertation we consider properties that use an intuitive notion of linear time, i.e. Linear Temporal Logic (LTL), which allows building logical formulas using the temporal operators such as ‘next’, ‘always’, ‘eventually’ and ‘until’. For example, when considering a fluctuating system, one could check that it stabilizes, i.e. that a certain state is ‘always eventually’ reached (Chapter 3). Importantly, stabilization is a liveness property: it says that ‘something good’ eventually happens. Another kind of properties are safety properties, which say that ‘nothing bad’ ever happens, for example that the concentration level of a protein is never low (Chapter 4). Both kinds of properties come up throughout this dissertation.
1.3 Goals and challenges

In this dissertation we investigate how to verify large discrete models of biological systems. Such systems typically consist of a large number of concurrently-executing small components. The main challenge when verifying them is state explosion, caused by the exponential nature of concurrent models. Namely, a system with \( N \) components, each in one of \( m \) possible states, has the total of \( m^N \) states. Enumerating states can easily exceed any finite amount of computer memory. Therefore, the main focus of this work is scalability: ability to handle systems with very large state spaces. In order to achieve scalability, we use existing techniques and propose new ones in the fields of high-performance computing and model checking.

The state explosion problem is approached in two ways in this dissertation. Chapters 2 and 3 use two techniques independent of state space enumeration, namely simulation and abstract interpretation (only executing parts of a program relevant to what we want to prove); these methods scale very well and are able to handle systems up to millions of variables. By contrast, Chapters 4 and 5 execute verification on a parallel computer, which can fit a larger state space; although eventually limited by the amount of memory available, this method enables implementing and applying general (Chapter 4) as well as custom (Chapter 5) verification methods.

1.4 Contributions

This dissertation makes the following contributions:

- **Chapter 2**: We develop a Petri-net model of vulva formation in the C. elegans worm; maximal parallelism is used for execution, which means that many transitions fire within a single step of the network. We check the model using Monte Carlo simulations: executing and analyzing a large population of in-silico animals with various genetic makeups. To make verification tractable, a linear approximation of maximal parallelism is created, and the experiments are parallelized for a cluster of computers. We found our model correctly emulates a large number of biological experiments, and new predictive experiments confirm published hypotheses about the stabilizing role of microRNA in the studied process.

- **Chapter 3**: We propose a novel scalable technique for proving stabilization (reaching a unique fixpoint) of large concurrent biological systems. Our tool interprets a program abstractly: it over-approximates the bounds of variables, and iteratively strengthens the bounds. It achieves scalability by applying state space exploration only locally to small pieces of the system rather than to the entire system as a whole. Using the new tool, we proved stabilization of a mesh of \( 200 \times 500 \times 5 \) cells of mammalian epidermis (skin); the state space of that model contains \( 2^{6 mln} \) reachable states.

- **Chapter 4**: We design and implement a framework for writing parallel graph algorithms that operate on large graphs split between memories
of multiple machines. The key idea is that a user can customize data and methods of graph vertices, and the framework provides seamless execution of methods on local and remote vertices, as well as handles the details of executing in a distributed environment. Using our framework, we implemented SPINJADI, a distributed on-the-fly enumerative model checker, a reimplementation of SPIN [22]. Using SPINJADI, we checked two mutual exclusion protocols, as well as a biological model of T-cell activation during an immune response.

- **Chapter 5**: We introduce an efficient distributed algorithm to find terminal strongly connected components (TSCCs) in large graphs. In biology, TSCCs correspond to states of terminal differentiation (when a cell stops specializing), or to steady states. Our algorithm is a parallel divide-and-conquer graph algorithm: using reachability computations, a graph is split into four independent subgraphs, which cannot be ‘crossed’ by SCCs, and so can be searched recursively in parallel. We found ours was the only algorithm able to process our case study: a model of human haematopoietic (blood) cells.

## 1.5 Outline of this dissertation

This dissertation is organized as follows. Generally, each chapter presents one method of verification of large concurrent systems, and applies it to one or more biological examples. Chapter 2 discusses Monte Carlo simulations of multi-cellular development in *C. elegans*. In Chapter 3 we describe a scalable technique to prove stabilization for concurrent systems. Chapter 4 introduces our framework to write distributed graph algorithms, as well as the distributed model checker created with it. In Chapter 5 we present a distributed algorithm to find TSCCs in large graphs. Chapter 6 concludes this dissertation. Additionally, the chapters are summarized in Table 1.1.

## 1.6 Collaborations

The work in this thesis was done mainly by me (at the VU University Amsterdam, Netherlands, and at Microsoft Research Cambridge, UK) with one exception. The work in Chapter 2 was performed together with Nicola Bonzanni (currently at the VU University Amsterdam and The Netherlands Cancer Institute): Nicola’s focus was on biological modeling; mine was on verification.

## 1.7 Bibliography


Table 1.1: Summary of the chapters in this dissertation. 'State space' is the size of the largest tested graph (for a graph $G = (V, E)$, it is $|V| + |E|$).

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Biological example</th>
<th>State space</th>
<th>Verification method</th>
<th>High-performance computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><em>C. elegans</em> vulval development</td>
<td>$2^{842}$</td>
<td>Monte Carlo simulation</td>
<td>Embarrassingly-parallel trace generation and grading</td>
</tr>
<tr>
<td>3</td>
<td>Mammalian dermis</td>
<td>$2^{6mln}$</td>
<td>Abstract interpretation; modular reasoning</td>
<td>Parallelization for a shared-memory machine</td>
</tr>
<tr>
<td>4</td>
<td>T-cell activation</td>
<td>$2^{34}$</td>
<td>Enumerative on-the-fly model checking</td>
<td>Parallelization for a cluster of machines</td>
</tr>
<tr>
<td>5</td>
<td>Haematopoietic cells</td>
<td>$2^{34}$</td>
<td>Search for terminal strongly connected components</td>
<td>Parallelization for a cluster of machines</td>
</tr>
</tbody>
</table>


Monte Carlo simulation of multi-cellular development

2.1 Introduction

Many efforts have been undertaken to elucidate how cells are able to coordinate different and sometimes conflicting signals, producing a precise phenotype during organ formation in animals. *C. elegans* vulval development [1] provides an elegant and relatively well-charted model to study the problem central to developmental biology: how multiple pathways, in multiple cells, interact to produce developmental patterns.

Defining suitable computational techniques for development modeling, able to perform *in-silico* experiments, is an open and challenging problem. We propose a coarse-grained quantitative approach based on Petri nets with an execution semantics adapted to realistically mimic biology. Namely, models are constructed as discrete bounded Petri nets, in which as many as possible concurrent components execute at a time—a principle called *maximal parallelism*. We apply the new approach to model fate determination during formation of a vulva, an egg-laying organ, in a small, soil-inhabiting, transparent worm, *C. elegans*. Our approach allows us to model synchronization, asynchronous events, conflicts and in general concurrent systems in a natural way.

In this chapter we study simulation-based verification—arguably the most commonly used verification technique [2]. The model of vulval development in *C. elegans* is verified using the Monte Carlo method, which consists in analyzing a large number of traces of a system. Indeed, for each given genetic perturbation, we performed 5000 executions of the model and matched the outcomes against the expected phenotypes. Using this technique, it is shown that our model correctly reproduces a large set of *in vivo* experiments, with statistical accuracy, as well as generates gene expression time series in accordance with biological evidence. In addition, we provide the necessary
statistical data to establish a more detailed comparison with biological observations than was previously possible.

Our model encodes different published hypotheses about the biology of vulval development \[3,4\]; notably, it includes the relatively recently discovered miR-61 microRNA (very short RNA that regulates protein synthesis). Using new \textit{in-silico} (in a computer) experiments to elucidate the involvement of miR-61 in down-regulating the lateral signaling, we confirm prediction \[4\] of its contribution in stabilizing cell pattern formation.

It would not be possible to efficiently handle the large model of multicellular organ formation in \textit{C. elegans}, was it not for the optimization and parallelization of the verification process. First, we optimized the Petri net execution by introducing a greedy approximation of the algorithm to generate a maximally parallel step; this reduced each step’s computation cost from exponential to linear with respect to the size of the network. Second, we parallelized the Monte Carlo verification for a cluster of computers, and executed it on 64 multicore machines. Combination of these two techniques reduced the time of verification, \textit{i.e.} a set of Monte Carlo experiments (one for each studied genetic perturbation), by many orders of magnitude: from the estimated 100 years to about one hour.

The remainder of this chapter is organized as follows. Section 2.2 contains an overview of the related work. In Section 2.3, we introduce our approach to modeling biological systems. Section 2.4 briefly explains the biology of vulval development in \textit{C. elegans}. We present the model that we built of the biological process in Section 2.5. The two next sections concern the verification method: Section 2.6 details the simulation procedure, whereas Section 2.7 shows how simulations were combined and parallelized as Monte Carlo experiments. We give results of verification in Section 2.8, followed by a discussion of the predictive experiments and biological results, in Section 2.9. Section 2.10 concludes this chapter.

This chapter has been published before, as:

and in parts as:
2.2. Related work

The first diagrammatic model, describing the regulatory network underlying cell fate determination during *C. elegans* vulval development, was proposed by Sternberg and Horvitz in 1989 [5]. Since then, global understanding of the biological network has improved greatly. The first computational model, proposed by Kam *et al* [6], combined multiple experimental 'scenarios' from [7] into a single model, using Live Sequence Charts (LSCs). Afterwards, in two landmark papers, Fisher *et al* [8, 9] suggested two state-based mechanistic models. The first [8] used statecharts to represent internal states of components, and LSCs to execute actions between them. They formalized Sternberg's model [5], but did not incorporate any additional data. A more recent approach [9] was based on Reactive Modules, with modeling principles akin to the previous paper. In contrast to the model presented in the current chapter, the three listed models build on representing rules that the system adheres to, rather than modeling the underlying biological processes.

Several other models of *C. elegans* vulval development have been published based on different modeling methods. Giurumescu *et al* [10] proposed a partial model based on ordinary differential equations (ODEs), while Sun and Hong [11] developed a model based on automatically learned dynamic Bayesian networks with discrete states. Independent from us, Li *et al* [12] modeled part of *C. elegans* vulval development using hybrid functional Petri nets with extensions. While they focused on model validation, we additionally generated new insightful predictions.

Petri nets are enriched in many ways in order to model biological systems [13–15]. For example, qualitative Petri nets [16] can be used for structural and invariant analysis, but they suffer from lack of precise biological data and from the assumptions similar to ODEs they must make. Stochastic Petri nets [17] incorporate kinetic constants, but these are mostly unknown or approximate. Hybrid Petri nets [18] and their extensions on which Cell Illustrator [14] is based, are rich and expressive, but model understanding and causal backtracking can be impeded by the complexity of the formalism. In this chapter we took a generally different approach: we did not enrich the formalism of Petri nets, but rather changed their execution semantic to mimic biology.

The maximally parallel execution semantic that we adapted was introduced by Burkhard [19]. A similar idea proposed by Fisher *et al* [20] is that of 'bounded asynchrony', in which a component can get ahead of another component by not more than *k* steps. In bounded asynchrony, components are more loosely synchronized than they are in maximal parallelism, which is more similar to a lockstep.

2.3 Biological interpretation of Petri nets

A Petri net [21, 22] is a bipartite directed graph that consists of two kinds of nodes: places, which indicate local availability of resources, and transitions, which describe dependencies between the resources (formal definition can be found at the end of this section). Each place can hold one or more tokens, which represent the number of resources available at the moment for example,
Monte Carlo simulation of multi-cellular development

in molecular biology, the resources include: genes, proteins, DNA, and other molecules. Weighted arcs connect places and transitions. When a transition fires it consumes tokens from input places, and produces tokens in output places. Firing of a transition is interpreted as execution of a biological event, for example creation of a complex molecule from two simpler molecules. Executing the entire network emulates the whole studied process, such as development of an organ.

Traditionally, Petri nets are discrete, unbounded, and execute fully asynchronously: in every step, one random transition fires. This basic formalism is often enriched in many ways to extend its expressiveness in order to model biological systems [13–15]. Instead of extending the formalism further (and its complexity), we preserve the simplicity of Petri nets, but we make the way they are executed biologically-relevant. Note that each component of the new semantics has been proposed earlier, but it is their combination and application to biological systems that is novel.

Discrete and bounded
In our approach, the number of tokens in each place is discrete and bounded. A gene is represented as a boolean value: one token means the gene is switched on, and no token means the gene is off. For proteins, we use abstract concentration levels 0–6: going from absence, via low, medium, and high concentration, to saturated level. This is in contrast with using the tokens to directly represent absolute numbers of molecules or fixed molar concentrations [16]. Since such exact quantitative data is often unavailable, we aim to represent relative concentration levels and qualitative information about dependencies between proteins.

Besides being discrete, the levels of proteins are also bounded, as unrestricted production of proteins is usually not realistic: in nature the cell would saturate with the product, and the reaction would slow down or stop. In addition, to guarantee that the highest concentration level can be attained, we introduce bounded execution with overshooting: a transition can fire if each output place has room for at least one token; the tokens above the maximum level are ignored. Note that the choice of seven concentration levels is somewhat arbitrary. We chose seven as we intend to stay in between a simple boolean level and a complex model of differential equations, and because seven concentration levels sufficed to express the biological knowledge about C. elegans vulval development in a satisfactory fashion. If required, the granularity of the model could be fine tuned by using more concentration levels, at the cost of a larger state space.

Maximal parallelism
Biological systems are highly concurrent, as in cells all reactions can happen independently in parallel; to emulate this in a Petri net, we apply a principle of maximal parallelism introduced by Burkhard [19]. A fully asynchronous approach would allow one part of the network to deploy prolonged activity, while another part of the network shows no activity at all. In real life all parts of a system progress at roughly the same rate [20], and so the key idea of maximal parallelism is to let as many transitions as possible fire
within a single step of the network. Formally, a parallel step is a multi-set of transitions; in particular, the same transition may execute more than once in a single step, provided there are sufficient input tokens and output slots for all these transitions to execute. A maximally parallel step is one that leaves no transition enabled in the network. When multiple maximally parallel steps are possible—as is often the case when transitions compete for a common resource—one maximally parallel step is selected at random.

For an example of maximally parallel steps, consider the network in Figure 2.1 (in this and later figures, if an arc weight is unspecified, it defaults to 1). In the figure, each of the transitions \( t_1, t_2, \) and \( t_3 \) is enabled twice in isolation, and they compete for common resources \( p_1 \) and \( p_2 \). When transition \( t_2 \) executes—twice, once, or not at all—the rest of the resources must be exhausted by \( t_1 \) and \( t_3 \). Therefore, the possible maximally parallel steps are: \( \{2 \times t_2\}, \{t_1, t_2, t_3\}, \) and \( \{2 \times t_1, 2 \times t_3\} \).

Note at this point that a maximally parallel step is potentially costly to compute, because there can be exponentially many—with respect to the number of components—possible maximally parallel steps. To see this, consider an example of \( n \) triangle-like structures in Figure 2.2. Each token must go either left or right; if we denote these events with 0 and 1, the set of possible maximally parallel steps corresponds to \( \{0, 1\}^n \), meaning \( 2^n \) steps.

Note also that, although maximal parallelism enforces activity throughout the network, the reactions within the network are able to proceed at different speeds. This is because the weights on arcs capture the relative rates of events and concentration levels, as corroborated by our experiments. Namely, if \( A \) is produced three times faster than \( B \), then the weight of the arc that produces \( A \) should be three times the weight of the arc that produces \( B \).

**Formal definitions**

A Petri net bounded by \( N \) is formally a 5-tuple: \( (P, T, F, W, M_0) \), where:

- \( P = \{p_1, p_2, \ldots, p_m\} \) is a finite set of places;
- \( T = \{t_1, t_2, \ldots, t_n\} \) is a finite set of transitions, such that \( T \cap P = \emptyset \);
- \( F \subseteq (P \times T) \cup (T \times P) \) is a finite set of arcs;
• $W: F \rightarrow \{0,1,\ldots\}$ is a weight function;

• $M_0: F \rightarrow \{0,1,\ldots,N\}$ is the initial marking.

In this chapter, we use $N = 6$.

Any function $M: F \rightarrow \{0,1,\ldots,N\}$ is called a marking. If $M_j(p) = k$, then we say that, at time $j$, the place $p$ contains $k$ tokens. Assuming a marking $M$, a transition $t$ is enabled if the two conditions hold:

• $M(p) \geq W(p,t)$ for each input place $p$ of $t$, and

• $M(r) < N$ for each output place $r$ of $t$ such that $W(t,r) > 0$.

A firing of a transition $t$ removes $W(p,t)$ tokens from each input place $p$ of $t$, and creates $\min(W(t,r), N - M(r))$ tokens in each output place $r$ of $t$.

Execution of a Petri net is a series of markings $M_0,M_1,M_2,\ldots$, such that each marking $M_{i+1}$ is generated from the previous marking $M_i$ by firing the transitions generated by the chosen execution semantics. In this chapter, we use the maximally parallel execution semantics. Consider marking $M$. A parallel step $s$ is $s: T \rightarrow \{0,1,\ldots\}$. A parallel step is enabled if, for each place $p$:

• the output transitions $t_1,t_2,\ldots$ of $p$ can consume:

$$M(p) \geq \sum_i W(p,t_i)s(t_i);$$

• the input transitions $t'_1,t'_2,\ldots$ of $p$ can produce:

$$M(p) + \sum_i W(t'_i,p)s(t'_i) \leq N,$$

or at most one of the transitions, $t_k$, ‘overshoots’:

$$M(p) + \sum_{i \text{ except } k} W(t'_i,p)s(t'_i) + W(t'_k,p)(s(t'_k) - 1) < N.$$

A maximally parallel step is a parallel step that leaves no transition enabled in the network. When multiple maximally parallel steps are possible, one is selected at random.

2.4 C. elegans vulval development

We used our new biological interpretation of Petri nets to model fate determination during vulva formation in the C. elegans worm. Our model is described in the next section, while here we briefly describe the biology underlying this process. We start by explaining the basic notions of molecular cell biology: if you know what DNA, proteins and transcription are you can safely skip the primer.
2.4. *C. elegans* vulval development

Cell biology primer

DNA is a long-term information storage in an organism; it is identical in all cells but not all cells behave the same as different pieces of DNA, called genes, can be active, or expressed, at different times. DNA governs cell behavior through proteins, which carry out all the labor in a cell, such as growth, replication, movement, or synthesis of new proteins. Biologists think about logical pieces of this complex machinery as pathways. Of particular interest are signaling pathways: loops and cascades of signals that function as a communication mechanism between cells, and within cells. For example the Ras/MAPK pathway, a major player in our model of *C. elegans* vulval development, is in general an important signaling pathway that transports extracellular growth signals received at the membrane into the nucleus, where they can be handled. Typically, signals are 'forwarded' by way of down- and up-regulation, i.e. when a quantity of a cellular component decreases or increases, respectively, upon influence of another component—an example of this is given in Section 2.5.

How are proteins obtained from DNA? The central dogma of molecular biology states that this happens in two phases: transcription followed by translation. During transcription, in the nucleus (if there is one, otherwise in cytoplasm), a gene is copied into a messenger RNA (mRNA) molecule, which then travels outside of the nucleus. In a ribosome, the cell's giant protein factory, the mRNA 'letters' are translated into a sequence of amino acids that form a protein. To take the computer science analogy suggested by Cohen [23], DNA is a stored program, which is 'executed' as a protein. One variation of this DNA-to-mRNA-to-protein process is that sometimes a cell uses RNA directly, without translating into protein. This is for example how microRNA works—in Section 2.9 we discuss the role of certain microRNA in vulval development of *C. elegans*.

Fate determination during vulval development in *C. elegans*

The vulva develops from six vulval precursor cells (VPCs), consecutively numbered P3.p to P8.p (see Figure 2.3). Each VPC is competent to respond to intercellular signals, and is potentially able to adopt either of the three cell fates: 1°, 2°, or 3°. The adopted fate determines the cell's participation in further development: the 1°s become the center of a vulva, the 2°s support the vulva, and the 3°s fuse with the worm's skin. In the wild-type worm (as in nature, i.e. not mutated) the VPCs invariably adopt the 3°-3°-2°-1°-2°-3° pattern [7], shown in Figure 2.3, which leads to a single vulva developing from the three middle cells P5.p-P7.p.

This precise wild-type fate pattern is the result of an interplay between two competing signals: the inductive signal produced by the Anchor Cell (AC) and the lateral signal initiated by the P6.p cell. First, the inductive epidermal growth-factor signal produced by the AC is transported to the three nearest precursor cells (depicted in Figure 2.3 with arrows from the AC to cells). This signal is strongest in P6.p, where it induces the 1° fate, inhibits the 2°
fate [3], and initiates lateral signaling [25] (horizontal arrows in Figure 2.3). The lateral signal in P5.p and P7.p promotes the 2° fate [26] and inhibits the 1° fate [27]. This negative feedback helps to reinforce different fates in neighboring cells [4]: namely, if a cell adopts the 1° fate, it coerces the flanking cells to adopt the 2° fate.

2.5 Modeling multi-cellular development

In this section we explain how we represented the process of cell fate determination during vulval development in C. elegans using the new approach of bounded maximally-parallel Petri nets (Section 2.3). The network that we built contains 300 places, 300 transitions, and 1000 arcs. It is displayed in Figure 2.4: although it is scaled down, one can distinguish the sub-networks that represent the anchor cell (AC), the worm’s skin (hyp7), and six interconnected VPC cells P3.p–P8.p. In addition, the models of individual VPCs are identical. Each cell contains eleven genes that guide the development, and eleven proteins that interact to arrive at the cell’s fate.

An example of representing biological knowledge as a Petri net is given in Figure 2.5(a). This small network represents protein production. Namely, the VAV-1 protein is synthesized if, and only if, the vav-1 gene is present. When the transition fires, it consumes one vav-1 token, and atomically produces one VAV-1 token and one vav-1 token (the number of tokens in vav-1 is constant). The protein production Petri net occurs often in our model, as it was used for all protein species in all cells.

Modularity

Protein production in Figure 2.5(a) is, in fact, an example of a functional module. Modules are sub-networks that represent logical pieces of the system; they can be used as building blocks to construct more complex modules, and eventually the entire system. Following this principle, we assembled our network in a five-stage bottom-up procedure. In the first stage, we identified six basic modules encoding 'atomic' biological functions: protein production discussed above, as well as protein activation, up-regulation, down-regulation, constitutive degradation, and signaling. In the next stage, we represented each
Figure 2.4: Schematic representation of the whole system. The VPCs are connected with the AC, hyp7, and the neighboring cells.
2.6 Simulation procedure

A simulation of our model is interpreted as an execution of an individual *C. elegans* worm, and the outcome of the simulation as the worm's phenotype. Since a maximally parallel execution is non-deterministic, to wholly verify the *C. elegans* model, a large population of worms, rather than a single worm, must be simulated for each studied genetic perturbation. Similarly, in experimental biology, replicating experiments is necessary to overcome the variability intrinsic to biological systems. Doing this *in silico* is called the *Monte Carlo* method, described in Section 2.7; in the current section we discuss the procedure to perform a *single* simulation. First, we explain how the simulation is initialized, *i.e.* the initial values for the participating genes and proteins. Second, the procedure to perform the steps of a simulation is described, in particular the reduction of its computational cost. Third, we
show how the worm’s phenotype is obtained from a finished simulation and checked against the biological data.

**Initialization: Genetic perturbations and calibration**

The most important input to our simulation is a genetic makeup for *C. elegans*, meaning the settings for all the participating genes. We consider three kinds of gene expression, or ‘strength’: (a) wild-type (*wt*), *i.e.* the expression level most common in nature, (b) a loss-of-function (*lf*), when the gene is deleted or dysfunctional, and (c) a gain-of-function (*gf*), when the gene is over-expressed. In our network, the genetic setup is simulated by placing tokens in predefined places that represent gene expression. Each combination of gene settings results in a correlated phenotype observed in a wet lab; some examples of gene combinations and phenotype fate patterns in *C. elegans* vulval development are listed in Table 2.1. Besides the genetic makeup, the concentration levels of proteins must be initialized before a simulation. In all experiments, we used the same settings for proteins obtained experimentally during model calibration.

**Execution: Approximate maximal parallelism**

Initially, performing a single simulation of our model took up to several hours on a desktop computer. Each simulation consists of 1000 maximally parallel steps—this number was chosen empirically, as sufficient for all our stabilizing experiments to reach a steady state. Executing a long simulation is costly, as each maximally parallel step involves choosing one element at random of an exponentially large set of possible steps (see Section 2.3). To speed it up, we created an efficient *greedy approximation* of maximal parallelism. In the approximated semantics, enabled transitions are taken at random until exhausted. During that procedure transitions consume tokens but do not output them; rather, after the enabled transitions are exhausted, all taken transitions produce tokens in one go, thus completing generation of the parallel step. Clearly, the cost of this procedure is linear with respect to the size of the network. Additionally, this procedure is sound, *i.e.* it produces a correct maximally parallel step, as well as it is able to produce any step that can be produced by the full semantics. However, the two semantics are not entirely equivalent: they differ in the distribution of probabilities for maximally parallel steps; for example, the probability of choosing the $\{2 \times t_2\}$ step in the example network in Figure 2.1 (see Section 2.3) is lower in the approximated semantics than it is in the original semantics. We experimentally found that both semantics produce similar results in the examples that we tested based on our model of *C. elegans* vulval development. Using the approximated semantics reduces simulation time to not more than three seconds for our model.

**Determining the phenotype**

After a simulation is finished, the fate of the VPC cells, and thus the worm’s phenotype, is determined by correlating the concentration levels of two proteins, MPK-1, a downstream product of the inductive signal, and LIN-12, which is activated by the lateral signal [3, 28]. Specifically, a high-level of MPK-1
induces the $1^{\circ}$ fate, independently of the level of LIN-12. If MPK-1 is low, the adopted fate is determined by LIN-12: high LIN-12 results in the $2^{\circ}$ fate, and low LIN-12 results in the $3^{\circ}$ fate. A dependence of fate on levels of (MPK-1, LIN-12) is displayed in Figure 2.6. If levels do not clearly fit any of these criteria, the result is non-deterministic: in nature, it would be determined by small variations in protein levels. We calculate the LIN-12 and MPK-1 concentration levels as the average number of tokens over the final 50 steps, in order to avoid unnecessary noise from the continual discrete changes in the number of tokens.

To each executed simulation we assign a score, a number in $[0, 1]$, which describes how well the reached cell pattern fits the expected pattern: 1 means a perfect fit, 0 means the results were far off, and scores between 0 and 1 are partial matches. The score for a phenotype is an average of scores for individual cells. A cell’s score is obtained by applying three sigmoid (i.e. S-shape) functions representing the three fates and choosing the fate that returns the highest score. In fact, the landscape in Figure 2.6 is a combination of the elevated regions in these three functions, similar to the piecewise function in [10]. Such a continuous score is more useful than a piecewise function to detect slight changes in scores when modifying the network, especially during calibration.

### 2.7 Monte Carlo verification

Monte Carlo verification relies on obtaining a large sample of traces of the model. For each studied genetic perturbation, we performed 5000 simulations with different random seeds. Each individual simulation was performed according to the procedure described in Section 2.6. A score of a Monte Carlo experiment was obtained as a distribution of scores of the individual simulations. Full verification of our model consists of 64 Monte Carlo experiments, each for one studied genetic perturbation. These experiments are described in Section 2.8, whereas in the current section we explain how the Monte Carlo experiments were carried out.
Parallelization
Initially, the entire suite of 5000 × 64 simulations, executed using the efficient approximated semantics (Section 2.6), would take more than a week to complete on a sequential machine. This is not acceptable, as verification must be performed multiple times during model calibration. To speed up the Monte Carlo experiments, we executed them on a parallel computer. The suite of Monte Carlo experiments consists of a large set of independent jobs of varying sizes. A single job is a sequential program, which reads the input model, applies the input perturbation, performs the non-deterministic simulation using an input seed for the random number generator, and outputs a score that describes how well the reached protein levels match the expected levels. The parallel application follows the master-slave model: each slave repeatedly requests a job from the master, executes it, and returns the result to the master, which computes the final score for each perturbation. Using 256 cores (64 machines × 4 cores) of the DAS-3 [29] cluster, the parallelization allowed the entire suite of in-silico Monte Carlo experiments to run in less than an hour.

2.8 Verification results

To determine the capability of our model to reproduce and predict the biological behavior of C. elegans vulval development, we simulated 64 different experimental conditions, which we partitioned into three sets. First, 22 experiments previously selected in [8] (Table 2.1) were used for calibration of the model. Second, 30 experiments were used for validation: 26 from [8], three from [1] (Table 2.2), and one from [4] (Exp. 52 in Table 2.4); in particular, experiment 51 (Table 2.2) was never simulated in any previous work that we are aware of. Third, the remaining 12 simulations constitute new predictions that invite untried experiments in vivo; the most remarkable insights (Table 2.4) are discussed in Section 2.9.

The main result of the verification of our C. elegans model is that it reliably reproduces all in vivo experiments, except for the double mutant lin-12(gf); lin-15(lf) (Table 2.1, Exp. 21 and 45), and even in these cases a fraction of the predictions matches the expected pattern. The noticeable differences of biological observations from different labs, and the few worms examined in vivo, do not help to establish a trustworthy expected outcome.

Unstable patterns
Of the 22 experiments in Table 2.1, particularly interesting are the experimental conditions that lead to unstable fate patterns, i.e. experiments 5, 21, 25, and 45. These results were already discussed in [9] and [11], but these discussions lacked statistical detail about the possible outcomes. In fact, [11] observed that the statecharts model of [8] often produces two adjacent 1°-fate cells, which they claim is rarely observed in experiments, but they also do not provide supplementary statistical details.

In Table 2.3 we provide statistical details for experiment 5. More than 93.4% of the predicted patterns match one of the expected biological 1°/2°-1°/2°-2°-1°-2°-1°/2° combinations. Of all matching patterns, only 4.5% contain
Table 2.1: A subset of in vivo experiments from [8] that we used for calibration of the model. In the anchor cell AC column, ‘−’ stands for no AC, while ‘+’ means that the AC is present. The Genotype column indicates mutation of genes: loss-of-function (lf) or gain-of-function (gf); if not specified, the gene is wild-type. By lst we denote the group of genes: lst-1, lst-2, lst-3, lst-4, and dpy-23; Vul groups let-23, sem-5, let-60, and mpk-1 genes. Cell fate such as 1°/2° means that the cell may adopt either of the two fates.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>AC</th>
<th>Genotype</th>
<th>Fate Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>lst</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vul</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lin-15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lin-12</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>g</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>c</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>h</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>d</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>c</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>c</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>c</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>c</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>d</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>c</td>
<td>−</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>c</td>
<td>−</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: In vivo experiments from [1] used for model validation.

<table>
<thead>
<tr>
<th>Exp.</th>
<th>AC</th>
<th>Genotype</th>
<th>Fate Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>let-60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lin-3</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>l</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>l</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>l</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

2.8. Verification results

Table 2.3: Statistical results for the Monte Carlo simulation of experiment 5 in Table 2.1 (lin-15(lf)).

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Fate Pattern</th>
<th>Occurrences</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wild-type patterns: 1° 2° 2° 1° 2° 1°</td>
<td>1180</td>
<td>23.6%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>946</td>
<td>19.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>830</td>
<td>16.6%</td>
</tr>
<tr>
<td></td>
<td>Three or more adjacent 2°-fate cells: 2° 2° 2° 2° 2° 2°</td>
<td>132</td>
<td>2.6%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>93</td>
<td>1.9%</td>
</tr>
<tr>
<td></td>
<td>Two adjacent 1°-fate cells: 1° 1° 2° 1° 2° 1°</td>
<td>88</td>
<td>1.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>46</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

Three or more adjacent 2°-fate cells, while just 2.7% have two or more adjacent 1°-fate cells. These quantities correspond to the biological evidence that in these experiments three adjacent 2°-fate, or two adjacent 1°-fate cells are very unlikely. In the remaining 6.8% of runs (not included in Table 2.3), one or more cells adopt the 3°-fate, and we interpret these outcomes as the ‘rare phenotypes’ in which uninterpretable lineages are observed (i.e. in between 2° and 3°), as noted for instance in [5].

Time courses

In our approach, each maximally parallel step corresponds to a step of the studied biological process. Thus the traces of the executed simulations can be interpreted as time courses of gene regulation in vulval development. In Figure 2.7, the gene expression time series generated by our model are compared against the fluorescent photomicrographs (photographs taken through a microscope) of vulval formation of a wild-type C. elegans published by [27]. In the photographs in Figure 2.7(a), the egl-17p::cfp-lacZ reporter is a detector for the inductive signal in the VPC cells. Figure 2.7(b) depicts the time series generated by our model in a simulation of the wild-type animal. Initially MPK-1 is faintly expressed in P5.p and P7.p. Subsequently, expression in P5.p and P7.p disappears, and MPK-1 remains at a high level only in P6.p, in accordance with the fluorescent photomicrographs of Figure 2.7(a). We note that the concentration levels at the end of the simulation are approximately constant, indicating a steady state. Additionally, in a related experiment [27], the lst genes were divided into two groups: pattern A which contains dpy-23 and lst-3, and pattern B to which lst-1, lst-2 and lst-4 belong. Each group has its own characteristic temporal expression pattern that corresponds closely to the time series generated by our simulation (not shown).
2.9 Biological results

Our computational model, besides reproducing well-known biological experiments, encodes and unifies different published hypotheses and conjectures, shedding light on vulval development process in *C. elegans*. This process is a result of an interplay between two pathways: the Ras/MAPK signaling of the inductive signal, and the LIN-12/Notch pathway governing the lateral signal, in which the LIN-12 protein is a major player [3, 27]. The two hypotheses described next are related to LIN-12 down-regulation, and a role of the miR-61 microRNA in vulval development.

**Encoded hypotheses**

Endocytosis is a process by which cells absorb big molecules by engulfing them. Shaye and Greenwald [3] propose that Ras activation leads to transcription of an unknown factor that enhances down-regulation of LIN-12 by altering its endocytic routing. In Figure 2.8 one can see how we captured this hypothesis in our model. Ras activation enables the transcription of the unknown gene, which down-regulates LIN-12 post-translationally. Notably, changing the model of LIN-12 down-regulation from post- to pre-translation disrupts this behavior and significantly alters our results.

Yoo and Greenwald [4] identified miR-61 as direct transcriptional target
of the Notch pathway. The *mir-61* gene encodes a microRNA which blocks expression of the mRNA encoding VAV-1—a protein involved in LIN-12 down-regulation—possibly promoting LIN-12 endocytosis. Therefore, they proposed that activation of miR-61 by LIN-12 and the consequent down-regulation of VAV-1 constitute a positive-feedback loop that promotes LIN-12 activity in presumptive 2°-fate VPCs. Although the unknown factor conjectured by Shaye and Greenwald does not seem to be required for the initial internalization of LIN-12, VAV-1 is necessary for the constitutive internalization of LIN-12. Notice that VAV-1 is involved in both constitutive and enhanced post-translation (endocytosis mediated) down-regulation of LIN-12.

**miR-61 as a developmental switch and modulator**

Modeling these hypotheses (Figure 2.8) and capturing their behavior has proved necessary to obtain the expected results during *in-silico* experiments. Moreover, we simulated several perturbations of the *mir-61* microRNA gene, obtaining the outcomes shown in Table 2.4. This nicely confirms the role of the positive-feedback loop proposed by [4]. All experiments of Table 2.4, as
far as we know, have not been tested in vivo (with the exception of experiment 52, which is described in [4]).

Experiments 52–55 confirm the specific role of miR-61 in influencing the cell fate decision, as determined by Yoo and Greenwald. Experiment 56 suggests a possible secondary role. This is a double mutant miR-61(lf);lst(lf) variation of the lst(lf) experiment 2, Table 2.1. Although the single mutant lst(lf) expresses a stable VPC fate pattern, the loss-of-function of miR-61 in the double mutant disrupts the stability of the pattern, as can be seen in the statistical breakdown of Table 2.5. Based on this observation, we suggest that besides acting as developmental switch, miR-61 plays a “tuning” role [38] to ensure the stability of the cell fate pattern formation.

To the best of our knowledge, we are the first to model in silico microRNA interactions during C. elegans vulval induction, supporting the conjecture formulated in [4] that lin-12, miR-61, and vav-1 form a feedback loop that helps maximize lin-12 activity in the presumptive 2°-fate VPCs.
In this chapter, we presented a method of modeling biological systems using bounded, discrete, maximally-parallel Petri nets. The new approach was applied to a multi-cellular process of fate determination during vulva formation in *C. elegans*, and yielded a large concurrent model of this process. We verified the model using Monte Carlo experiments, *i.e.* by obtaining and analyzing a large number of traces of the system, and we found we could reliably simulate a large set of *in vivo* experiments. Importantly, statistical accuracy of worm populations was achieved, as well as an agreement of the time series with photomicrographs. In order to make it tractable, the verification was optimized and parallelized for a cluster of machines.

The network that we created is fairly large: roughly 70 genes and 230 proteins, which result in a state space of size $2^{70} \times 7^{230}$, meaning an order of $2^{715}$ of different possible states of the system. Clearly, verification by way of enumeration of states is not viable. Instead, given a genetic makeup, we described the range of the worm’s typical behaviors by analyzing an *in silico* population of animals—obtained inexpensively and ethically—undergoing the studied biological process. This method delivers results fast, scales well, can be easily parallelized, and is applicable to a wide class of systems and properties. Nevertheless, it has two downsides. First, it offers no guarantees on coverage: one only attempts to cover the typical behaviors, rather than all possible series of events. Second, it does not help much in deciphering the control flow in the studied system, *i.e.* causality—development of tools to support inferring causality from simulations is an important point in future work. In spite of these two limitations, for those wanting to verify large non-deterministic systems, simulations are typically a go-to method, or at least a method applied to get the first understanding of the studied biological model.

### Bibliography

Monte Carlo simulation of multi-cellular development


Biologists are increasingly turning to computer science techniques, in particular to formal verification, in the quest to understand and predict the behavior of complex biological systems. Unlike simulation-based methods (Chapter 2), in formal verification biological models are analyzed as computer programs, which allows rigorous checking of all states or all paths of a model. In some cases known formal verification techniques work well, e.g. [1–4]; unfortunately, in other cases—such as proving stabilization [5]—we find existing abstractions and heuristics to be ineffective.

In this chapter we address the open challenge of finding scalable algorithms for proving stabilization of biological systems. We define stabilization as the existence of a unique fixpoint state that is always eventually reached. A proof of stabilization elucidates system robustness with respect to time, while stabilization counter-examples give insight into system homeostasis (stability) imbalance—in all cases the results are useful to biologists. In general, systems are expected to stabilize, particularly in developmental biology, but for some systems, such as the heart beat or the circadian (day/night) clock, cyclic behavior rather than stability is the desired outcome. In fact, note that a biologist checking stabilization of a system, always knows if the system must stabilize or must not stabilize. If the result conflicts the underlying biology, there must be a bug in the model, or a gap in understanding the biology of the studied system.

The solution to proving stabilization presented in this chapter bases on abstract interpretation, a technique proposed by Cousot and Cousot [6]. In abstract interpretation, a property about a program is proved by interpreting the parts of the program relevant to the property at hand, and ignoring the parts not relevant to that property. In other words, programs are executed
partially, in such a way that abstract computations give information on the actual computations, without performing them. Abstract interpretation can speed up the verification, as well as significantly reduce the amount of memory it needs. An intuitive example of abstract interpretation is ‘the rule of signs’, e.g. determining the sign of the result of multiplication from the signs of the multiplicands; another example is static code analysis commonly carried out by development tools, e.g. checking that each used variable is declared earlier in the code.

Abstract interpretation of a large concurrent program can still suffer from the state explosion problem, which forces us to use some form of modular reasoning, i.e. being able to compose specifications of components that execute concurrently. Since stabilization is formally a liveness property, we must be careful when using the powerful cyclic modular proof rules (e.g. [7,8]), as they are only sound in the context of safety [9]. Furthermore, we find that the complex temporal interactions between the modules are crucial to the stabilization of the system as a whole; meaning that we cannot use scalable techniques that simply abstract away the interactions altogether.

The biological systems considered in this chapter consist of concurrent communicating components modeled in the Qualitative Networks (QNs) formalism [10] or Regulatory Networks (RNs) [11]. In both formalisms, each component is a variable in \{0, 1, \ldots, N\} where \(N\) is a small fixed constant. Variables (in this chapter, we say ‘component’ and ‘variable’ interchangeably) are updated in increments or decrements of 1, depending on the values of input components (the exact formula is explained in Section 3.3). In QNs, all variables are updated synchronously in lockstep, while in RNs updates are asynchronous.

In this chapter we present a procedure for proving stabilization of biological systems modeled as QNs or RNs. The key idea is that we break the global liveness property (stabilization) into a chain of small liveness properties, called local lemmas. Each local lemma describes a small part of the system in isolation, and can thus be solved using quick local proof techniques. We must now answer two questions: which form of lemmas should we use; and how to find the lemmas that imply stabilization.

The key to our tool’s performance is the observation that it suffices to take the lemmas only of a very limited form of a liveness property about a single component:

\[
[FG(p_1) \land \ldots \land FG(p_k)] \Rightarrow FG(q),
\]

where \(p_1, \ldots, p_k\) are atomic formulae over inputs of a small component that we want to reason about, \(q\) is an atomic formula about this component’s output, \(F\) denotes ‘eventually’ in LTL [12], and \(G\) denotes ‘always’. We compute the set of all provable lemmas of this form by iterative strengthening; namely, we walk the structure of the system and strengthen knowledge about individual variables, in particular the lower and upper bounds on variables. After this procedure, if for each component \(v\) its lemma implies \(FG(v = k_v)\) for some constant \(k_v\), we have proved stabilization. Otherwise, we search for a counter-example to stabilization using the lemmas to restrict the counter-example search space.
3.1. Introduction

Our stabilization proving procedure is sound and complete. We experimentally confirm that it is scalable. We find that our lemma generation procedure accelerates both the proving as well as the disproving of stabilization. Section 3.7 demonstrates with experimental evidence how our lemma generation procedure leads to many orders of magnitude speedup in cases where known previous techniques work, and new results in cases where known techniques fail. These include challenging published examples such as: a 3-D model of the mammalian epidermis (skin) based on [10]; a model of metabolic networks operating in type-2 diabetes [13]; a model of fate determination of vulval precursor cells in C. elegans [14]; and a model of pair-rule cross-regulation during segmentation in Drosophila fly embryo [15]. Applying our procedure to the multidimensional model of epidermis revealed a bug in the model from [10], as we proved the system non-stabilizing. Consulting the biological papers corroborated that the model was, in fact, in disagreement with the biological evidence. After fixing the bug we could prove the system stabilizing (see Section 3.3).

Our algorithm depends on a domain $L$ over which lemmas range. In systems encoded as Qualitative Networks and Regulatory Networks, the discrete and bounded components are updated in increments or decrements of 1, meaning it is sufficient to consider in $L$ lemmas that restrict variables to one subrange. This insight is the basis for an optimization of the lemma generation algorithm, which works extremely fast in practice. Using the optimized algorithm we were able to verify systems with up to three million of discrete variables (see Section 3.7). When considering this optimization, our technique can be thought of as analyzing the system using abstract interpretation over the interval domain [6].

Our technique is geared towards efficiently proving stabilization where the proof can be teased out by examining the system’s compositional structure. This lemma generation strategy is potentially an overhead that may hinder rather than help performance in some cases. Namely, this happens for non-stabilizing systems, for which a counter-example can be found fast without bounding the search space. In Section 3.7 we demonstrate an example of this.

An advantage of our procedure is compositionality: the local stabilization lemmas give a specification that, when established for new components, implies the whole system’s stabilization without re-running the entire procedure. This can lead to experimenting with alternative components (e.g. testing modified components during a search for new drugs). This observation also leads to a practical advantage, as we check lemmas in parallel during the proof search.

The remainder of this chapter is organized as follows. Section 3.2 contains an overview of the related work. In Section 3.3, we introduce the problem under consideration and our solution to it, explained informally using a real-life biological example: a model of mammalian epidermis. Section 3.4 gives the formal definitions, and Section 3.5 the full algorithm, including the optimizations and counter-example search. In Section 3.6, we present formal proofs of soundness and completeness of the algorithms in Section 3.5. The experimental results are discussed Section 3.7. We conclude with Section 3.8.
3.2 Related work

With the exception of [10], no tools have been previously reported that are directly tailored to the problem of proving stabilization or other liveness properties of large biological systems modeled as discrete systems (e.g. Qualitative Networks). Classic theory of stability of differential equations is applied to continuous systems, e.g. in [16]. Recent work is known on the stability of hybrid systems, e.g. [17–19]. In the context of stabilization for discrete systems, Schaub et al [10] use the compositional structure of a system modeled as a Qualitative Network to accelerate the computation of a fixpoint-based computation of the reachable states. However, the final check is not modular, and thus is less scalable than our approach. Regulatory Networks [11], which are an ‘asynchronous’ variant of Qualitative Networks, have been extensively studied, e.g. in [11, 15], but the analysis relies on state space enumeration, which is not scalable, or stable states computation that does not account for reachability [20].

The current state-of-the-art amongst biologists interested in stabilization is to use either techniques from [10] or off-the-shelf model checking tools for finite-state systems. Recently developed tools for proving liveness of infinite-state systems (e.g. [21]) could also be used. Abstract interpretation was also used in a way similar to our method in [22] to produce tail invariants for termination proofs. As we show in Section 3.7, our procedure is many orders of magnitude faster than previously known approaches. The challenge is that biological models are very large, causing timeouts and out-of-memory failures for most tools not based on modular proof strategies. Note also that stabilization is not directly expressible in popular temporal logics, e.g. CTL or LTL, unless support for quantifiers is added, making the encoding of stabilization tricky in most formal verification tools. Qualitative Networks could be implemented in Lustre [23], which however only supports checking safety properties.

We are not the first to attempt to address the difficulty of modular reasoning for liveness. For example, several previous papers have reported on heuristics tailored to the problem of proving liveness of non-blocking concurrent programs [21, 24]. Their motivation is the same as here, but the approaches used differ as they are tailored to different problems. Another technique, as found in [25], is to use induction over time to facilitate the
modular proving of liveness properties of finite-state models. In [25] the modular decomposition is given manually, whereas in our work we use the structure of the biological system to our advantage when automating the search for the modular decomposition. To show that our proofs are non-circular we use an argument similar to that of [25].

### 3.3 Example: Skin cells

Figure 3.1 contains a pictorial view of a simplified model of mammalian epidermis (outermost skin layer) that consists of five stacked cells [10]. Each cell represents a single skin layer and communicates with neighboring cells. The bottommost cells proliferate, migrate upwards and eventually decide to die and thus contribute to the cornified skin surface. It is this balance between proliferation and cell death that makes the system interesting to biologists: too much death is detrimental to the skin, too little is cancerous. The original model is expressed as a Qualitative Network [10]; formal definitions of Qualitative Networks and Regulatory Networks are given later, here we describe the epidermis model only informally.

The example model includes a few executing components, each updating a single variable. See, for instance, the \( wnt_3 \) and \( \text{NotchIC}_3 \) variables in Figure 3.1. Each variable holds a value, which is a number in \( \{0, 1, \ldots, N\} \), where \( N + 1 \) is a predefined, globally-fixed granularity. A target function, \( T_v \), associated with each variable, \( v \), determines how the variable is updated, depending on the current evaluation of the target function:

\[
v' := \begin{cases} 
  v + 1 & v < T_v, \\
  v - 1 & v > T_v, \\
  v & v = T_v.
\end{cases}
\]  

(3.1)

In a Qualitative Network all variables are updated synchronously in parallel, whereas in a Regulatory Network they are updated asynchronously.

Intuitively, the update function of each variable is designed such that the value of the variable follows its target, which depends on other variables. In the biological setting, the typical target of a variable, \( v \), combines the positive influence of variables \( w_1, w_2, \ldots, w_s \) with the negative influence of variables \( w_{s+1}, w_{s+2}, \ldots, w_{s+r} \), and ignores all other variables in the network:

\[
T_v(w_1, w_2, \ldots, w_{s+r}) = \max \left( 0, \left[ \frac{1}{s} \sum_{k=1}^{s} w_k - \frac{1}{r} \sum_{k=1}^{r} w_{s+k} \right] \right)
\]

Graphically, this is often represented as an influence graph with \( \rightarrow \) edges between each of \( w_1, w_2, \ldots, w_s \) and \( v \), and \( \leftarrow \) edges between each of \( w_{s+1}, w_{s+2}, \ldots, w_{s+r} \) and \( v \). In this section we discuss only several target functions used in the skin example; papers [10,13–15] contain target functions used to model a large spectrum of aspects of signaling pathways, metabolic and regulatory networks.

Briefly, the target functions of variables in the skin models are as follows. The target of \( wnt_3 \) is \( T_{wnt_3} = N - \text{NotchIC}_3 \), which means that \( \text{NotchIC}_3 \)
Figure 3.1: Pictorial view of the skin model (rightmost cell is at skin surface). The bubbles show the underlying update functions for several of the variables in the model.
3.3. Example: Skin cells

inhibits wnt\(_3\) (in Figure 3.1 this fact is indicated by a 'blocking' arrow from NotchIC\(_3\) to wnt\(_3\)). The target of NotchIC\(_3\) is \(T_{\text{NotchIC}_3} = \min(3, \text{deltaext}_3)\) (indicated in the figure by an underline). The targets of the ext-variables round averaged cell inputs, which effectively requires at least one of the components to be present for some event to take place:

\[
T_{\text{deltaext}_1} = \left\lceil \frac{\text{delta}_0 + \text{delta}_2}{2} \right\rceil, \quad T_{\text{wntext}_1} = \left\lfloor \frac{\text{wnt}_0 + \text{wnt}_2}{2} \right\rfloor.
\]

Figure 3.1 shows behavior of four selected variables, based on their target function.

To complete the story about the biology of mammalian epidermis, the cell’s fates are determined by the levels of NotchIC and delta in the stable state (stability is discussed next in this section). Namely, in the \(k\)-th cell: if NotchIC\(_k\) > delta\(_k\), the cell terminally differentiates; if NotchIC\(_k\) < delta\(_k\), the cell proliferates; if the values are equal the cell is in transition. Generally, in a healthy skin, the bottom cell (cell with Notch\(_0\), leftmost in Figure 3.1) proliferates, the cell above is in transition, and the cells in the three top layers are terminally differentiated.

**Stabilization**

If all executions end in the same cycle, and that cycle has length 1, then we say the network stabilizes. Note that both Qualitative and Regulatory Networks are finite-state systems with only infinite executions. Thus, every execution must eventually end in some type of cycle. Stabilization guarantees both that the system has only a single fixpoint and that the fixpoint is always eventually reached—a violation of this property is the existence of two fixpoints or a cycle of length greater than 1. Biologists are often interested to see what this fixpoint is when it exists, and to see a counter-example when it does not.

When applied to the skin example, our tool incrementally finds a modular proof of stabilization, as depicted in Figure 3.2. The tool starts by guessing simple facts of the form \(FG(p)\) about variables that can be proved locally, i.e. using the update function of only one variable, with the definitions of the other variables abstracted away, see Figure 3.2(a). In this case, we can infer locally the lemma \(FG(\text{deltaext}_4 > 0)\) in the top cell. This property is provable using only local reasoning; namely, the deltaext\(_4\) variable follows a target \(\left\lceil (2 + \text{delta}_3)/2 \right\rceil\), which is always a positive number, independent of the value of delta\(_3\).

In the next step, we iteratively use the established facts to guide the search for additional facts to conclude. We search for locally provable facts of the form \(FG(p) \Rightarrow FG(q)\), where we only try to prove \(FG(p) \Rightarrow FG(q)\) if \(FG(p)\) is a consequent in a previous iteration. To continue our example, in Figure 3.2(b) we locally infer that

\[
FG(\text{deltaext}_4 > 0) \Rightarrow FG(\text{NotchIC}_4 > 0).
\]

This implication holds as \(\min(3, \text{deltaext}_4)\), the target of NotchIC\(_4\) in the top cell, effectively equals deltaext\(_4\), because \(N = 3\) is the maximal possible value of variables; since deltaext\(_4\) is eventually always positive, so is NotchIC\(_4\), up to \(N\) steps later.
Figure 3.2: Proof steps of skin model stabilization. Each arrow denotes a lemma.
In the next round, we can prove

\[ FG(\text{NotchIC}_4 > 0) \Rightarrow FG(\text{wnt}_4 < N) \]

in the top cell, see Figure 3.2(c). This property holds locally, because the target of \( \text{wnt}_4 \) is \( N - \text{NotchIC}_4 \). Figure 3.2(c) also contains several subsequent stages of the proof. We continue such reasoning until no new implications can be deduced. At that point, if we can conclude \( \ldots FG(v = k_v) \) for some \( k_v \in \{0, 1 \ldots N\} \) for each variable \( v \), then we have found a global stable state and proved that the model stabilizes.

**A bug in the skin model**

Applying our tool to the 1-D skin model described above proved the model stabilizing. By contrast, applying the tool to the 2-D skin model built out of several interconnected such 1-D models, proved the 2-D model not stabilizing. This result is biologically surprising, so we suspected a bug in the original model from [10]. After consulting biological literature [26], we discovered that the bug was real, i.e. the original model was in disagreement with biological evidence. The fix proposed was to change the value of the Notch protein (constant input of NotchIC\(_0\) in the bottommost skin layer) from 0 to 1. By doing so we effectively introduced a low level of Notch protein into the basal layer of epidermis. With the bug fixed, we proved the multidimensional model stabilizing. While this finding offered no new biological insight, it helped to repair the existing model and confirmed the usefulness of our method to biologists.

### 3.4 Preliminaries

**Qualitative Networks (QN)**

Following [10], a **Qualitative Network** (QN), \( Q(V, T, N, n) \), of granularity \( N+1 \) consists of \( n \) variables: \( V = (v_1, v_2 \ldots v_n) \). The state of the system is a finite map \( s: V \rightarrow \{0,1,\ldots N\} \). The initial state is random. Each variable \( v_i \in V \) has a **target function** \( T_i \in T \) associated with it: \( T_i: \{0,1,\ldots N\}^n \rightarrow \{0,1,\ldots N\} \). Qualitative Networks update the variables using synchronous parallelism. Target functions direct the execution of the network: namely, from a state \( v = (v_1, v_2 \ldots v_n) \), the **next state** \( v' = (v'_1, v'_2 \ldots v'_n) \) is computed by:

\[
    v'_i = \begin{cases} 
    v_i + 1 & v_i < T_i(v), \\
    v_i & v_i = T_i(v), \\
    v_i - 1 & v_i > T_i(v). 
    \end{cases} \tag{3.2}
\]

**Regulatory Networks (RNs)**

A **Regulatory Network** (RN) [11], \( G(V, T, N, n) \), consists of \( n \) discrete variables: \( V = (v_1, v_2 \ldots v_n) \) bounded individually by \( N : V \rightarrow \{1,\ldots, N\} \). Variables have target functions from \( T \) associated with them that govern updates of variables, as in formula (3.2). The updates are asynchronous, which is the main difference between RNs and QNs. We additionally assume that
the updates are fair, *i.e.* each variable that is not equal to its target value is eventually updated.

**Biological interpretation of QNs and RNs**

QNs and RNs have proven to be a suitable formalism to model biological systems [10, 11, 13–15]. A target function of a variable \( v \) is typically a simple algebraic function, such as sum, over several other variables \( w_1, w_2 \ldots w_m \). We often say that \( v \) *depends* on \( w_1, w_2 \ldots w_m \) or that \( w_1, w_2 \ldots w_m \) are *inputs* of \( v \). \( Q|_v \) denotes the restriction (sub-network) of the network \( Q \) to the variable \( v \) and its inputs, where the inputs behave arbitrarily (have no inputs of their own). In the following, we use the term *network* to refer to both QNs and RNs.

**Stabilization**

We say that a network is *stabilizing* if there exists a unique state \( s \) that is eventually reached in all executions, such that \( T(s) = s \). Intuitively, this means that, when walking the state space from an arbitrary state, or when starting a simulation in an arbitrary state, we always end up in a fixpoint and always the same one. Formally, we are attempting to prove the existence of a unique state \((k_1, k_2, \ldots k_n)\) such that \( FG(\forall v_i \in V. \ v_i = k_i) \). Note that the stabilization property is not expressible in LTL unless we add support for both existential and universal quantification over states.

### 3.5 Stabilization algorithm

In this section we describe our algorithm, which attempts to efficiently prove stabilization of systems using the modular strategy exemplified in Section 3.3.

Since the networks considered are finite and all executions are infinite, each execution of the system must end in a cycle. We consider all possible executions of a network and note the trichotomy: (a) all executions end in the same fixpoint (the network stabilizes); or (b) there exists an execution that ends in a cycle of length greater than 1 (the network cycles); or (c) all executions end in a fixpoint and there exist at least two different fixpoints (the network bifurcates). As described later in this section, our algorithm covers all of these cases, and is therefore complete. We note that completeness depends on the finiteness of networks considered and on the fact that the algorithm falls back on the non-compositional CexSearch routine.

**Notation**

Given a network \( Q \) with a set of variables \( V \), we define \( L \) to be a finite set of predicates that range over the simple inequalities of the form \((m \leq v \leq M)\), where \( v \) is any variable in \( V \), and \( m \) and \( M \) are constants in \( \{0, 1, \ldots, N\} \).

We use the term *local lemma* over a variable \( v \) to represent proved assertions of the form

\[
FG(p_1) \land FG(p_2) \land \cdots \land FG(p_m) \Rightarrow FG(q),
\]

where \( q \in L \) restricts \( v \), and \( p_1 \) through \( p_m \) are predicates about variables in the network proved previously.
Algorithm 3.1: Stabilization proving procedure for a network $Q$ with a set of variables $V$ bounded by $N$. If $Q$ is stabilizing, returns the fixpoint; otherwise, returns a counter-example.

```plaintext
procedure PROVE($Q : \text{Network}(V, N)$)
returns Fixpoint or Counterexample
1: $(V^{\text{min}}, V^{\text{max}}) := \text{GENLEMMAS}(Q)$
2: if $(\forall v \in V. V^{\text{min}}(v) = V^{\text{max}}(v))$ then
   3: return Stabilizing at fixpoint $V^{\text{min}}$
else if CEXSEARCH($V^{\text{min}}, V^{\text{max}}, Q$) finds a counter-example $\pi$ then
5: return Non-stabilizing with counter-example $\pi$
else
7: $\rho := \text{Single fixpoint of } Q$
8: return Stabilizing at fixpoint $\rho$
9: end if
```

The stabilization algorithm (Algorithm 3.1, PROVE)

Our procedure is displayed in Algorithm 3.1. It first applies a local lemma generation procedure GENLEMMAS (Algorithm 3.2) that is explained next in this section. In all practical cases we find that the lemmas found during this phase directly imply stabilization in cases where the model does stabilize. If no proof has been found, the strategy is reversed: our procedure searches for one of two types of counter-examples: multiple fixpoints and non-trivial cycles. Both counter-example finding procedures are complete; therefore, in the instance that GENLEMMAS does not prove stabilization and yet no counter-example is found, we have still proved stabilization. The procedure CEXSEARCH($V^{\text{min}}, V^{\text{max}}, Q$) is used by Algorithm 3.1 to look for existence of a counter-example in a network $Q$. Importantly, CEXSEARCH uses the proved variable constraints $V^{\text{min}}$ and $V^{\text{max}}$ to reduce the state space it needs to explore. If CEXSEARCH is unable to find a counter-example, no counter-example exists. Thus, in this case, we know that we only need to find a single trivial cycle. This is easily done using a decision procedure as in CEXSEARCH, or by simulating the system from an arbitrary state until the fixpoint is found.

Lemma generation (Algorithm 3.2, GENLEMNAS)

The key idea behind our approach is to first find local lemmas about the update functions for specific variables in the network. That is, if a variable $v$ locally depends on variables $w_1, w_2, \ldots, w_m$, we compute lemmas about interactions between $v$ and $w_i$'s of the following form:

$$FG(p_1) \land FG(p_2) \land \cdots \land FG(p_m) \Rightarrow FG(q)$$

where $p_i$'s are predicates in $L$ about variables $w_i$'s, and $q$ is a predicate about $v$. We compute local lemmas until no new ones can be deduced. If for each variable $v \in V$ we can use the lemmas to prove that $FG(v = k_v)$ for some constant $k_v$, then we can report that the system is stabilizing.
Algorithm 3.2: Lemma generation procedure GENLEMMAS for a network $Q$ with a set of variables $V$ bounded by $N$. Returns the set of all provable local lemmas about variables in $V$; these imply the bounds on variables $V^\text{min}, V^\text{max}$.

```plaintext
procedure GENLEMMAS($Q : \text{Network}(V, N)$) returns $V^\text{min}, V^\text{max}: V \rightarrow \{0, 1, \ldots, N\}$
1: $\mathcal{F} := \emptyset$; $\mathcal{P} := \emptyset$
2: $\forall v \in V, v \text{ constant}$. $V^\text{min}(v) := v \land V^\text{max}(v) := v$
3: $\forall v \in V, v \text{ non-constant}$. $V^\text{min}(v) := 0 \land V^\text{max}(v) := N$
4: for all non-constant variable $v \in Q$ do
5:     $\mathcal{F} := \mathcal{F} \cup \{v\}$
6: end for
7: while $\mathcal{F} \neq \emptyset$ do
8:     $w := \text{pick a variable from } \mathcal{F}$
9:     for all variable $v \in \text{outputs}(w)$ do
10:        for all lemma $l \in \text{NEWLEMMAS}(v, V^\text{min}, V^\text{max})$ do
11:            $\mathcal{F} := \mathcal{F} \cup \{v\}$
12:            $\mathcal{P} := \mathcal{P} \cup \{l\}$
13:            update bounds $V^\text{min}(v)$ and $V^\text{max}(v)$ as implied by $l$
14:        end for
15:     end for
16: end while
17: return $(V^\text{min}, V^\text{max})$
```

The procedure GENLEMMAS, displayed in Algorithm 3.2, iteratively computes a set of lemmas, $\mathcal{P}$. During the iterative search, it maintains a set of frontier variables, $\mathcal{F}$, for which new facts have been proved, but not used yet. Initially, $\mathcal{F}$ contains all unfixed variables in the network (Lines 4–6). The procedure repeatedly picks variable $w$ from $\mathcal{F}$ (Line 8), and generates new local lemmas about variables that depend on $w$ (Lines 9–10). The new lemmas are used to strengthen $V^\text{min}$ and $V^\text{max}$ (Line 13), the bounds that over-approximate the variables; namely, for each $v \in V$ we have

$$FG \left( V^\text{min}(v) \leq v \leq V^\text{max}(v) \right).$$

Algorithm 3.2 terminates because a variable's bounds can be updated at most $N$ times, so each variable can be enqueued at most $N$ times. From this it also follows that GENLEMMAS performs no exponential explorations. Generation of the new local lemmas NEWLEMMAS is shown in Algorithm 3.3. For a given variable $v$, it searches the language of base inequalities $\mathcal{L}$ for predicates about $v$ that improve its current approximation; it checks the lemmas and returns those that are true. The cost of NEWLEMNAS depends on the size of $\mathcal{L}$; since $\mathcal{L}$ is a language of simple inequalities over each variable, the loop in Line 3 executes at most $(N + 1)^2$ times, where $N$ is usually a small constant in biology. Therefore, the computational cost of NEWLEMNAS is $O(n + (N + 1)^2) = O(n)$, assuming a constant cost for PROVELEMA (see the
Algorithm 3.3: Procedure NEWLEMMAS that generates new lemmas about a variable $v$ in a network with variables $V$ bounded by $V_{\text{min}}$ and $V_{\text{max}}$.

**procedure NEWLEMMAS**($v \in V; V_{\text{min}}, V_{\text{max}}: V \rightarrow \{0, 1, \ldots N\}$)  
returns Lemmas

1: $S := \emptyset$; $(w_1, w_2, \ldots, w_m) := \text{inputs}(v)$  
2: $p := (\bigwedge \forall i V_{\text{min}}(w_i) \leq w_i \leq V_{\text{max}}(w_i))$  
3: for all predicate $q \in L$ over $v$ that strengthens $V_{\text{min}}(v)$ or $V_{\text{max}}(v)$ do  
4: $l := (FG(p) \Rightarrow FG(q))$  
5: if PROVELEMMA($l, Q|_v$) then  
6: $S := S \cup \{l\}$  
7: end if  
8: end for  
9: return $S$

following discussion). The worst-case complexity of GENLEMMAS is $O(n^2 N \cdot n) = O(n^3)$.

Recall that with $Q|_v$ we denote the restriction of $Q$ to a variable $v$ and its inputs. The call PROVELEMMA ($\phi, Q|_v$) is the application of model checking techniques to prove that $Q|_v$ respects the property $\phi$; we used the Cadence SMV model checker [27] (see Section 3.7). The key to the performance of our implementation is that checking $\phi$ locally is extremely fast. Since we are able to prove stabilization of the entire system while only ever applying PROVELEMMA to small parts of the system, our procedure is very efficient. That, coupled with the fact that PROVELEMMA calls can be executed in batches and thus in parallel on as many processors as are available, makes the method scalable.

Theorem 3.1 establishes the soundness and completeness of our method.

**Domain specific optimization**

Until now we have presented a general procedure that works with most models of concurrent updates, and all possible update relations (not just those defined per variable to follow the target functions). However, due
to specific target functions used in QNs and RNs, we can reimplement the lemma generation routine in such a way that PROVELEMMA is never needed, leading to significant performance improvements. Our alternative procedure F-NEWLEMMA is shown in Algorithm 3.4. We consider a variable \( v \) and its inputs \( w_1, w_2, \ldots, w_m \). Instead of guessing the influence of inputs under the constraints \( V_{\text{min}} \) and \( V_{\text{max}} \) on the output \( v \), we compute it exactly. Namely, we compute the set \( T \) of values of target function \( T_v \) applied to all possible input combinations:

\[
T = T_v \left( [V_{\text{min}}(w_1), V_{\text{max}}(w_1)] \times [V_{\text{min}}(w_2), V_{\text{max}}(w_2)] \times \ldots \times [V_{\text{min}}(w_m), V_{\text{max}}(w_m)] \right), \tag{3.3}
\]

thus obtaining a new approximation for \( v \): \( \min(T) \leq v \leq \max(T) \). In Theorem 3.2 we argue the correctness of this procedure, i.e. that the lemmas generated by F-NEWLEMMA indeed hold.

The worst-case cost of the stabilization proving procedure using F-NEWLEMMA is \( O(n^2 N_d + 1) \) where the network has \( n \) components, of maximal in-degree \( d \) (\( N_d \) results from generating input combinations). Since in all of our examples \( N \) and \( d \) are small, this procedure works exceptionally fast (see the experimental results in Section 3.7). If \( N \) or \( d \) were large, the procedure with NEWLEMMA could in principle be more efficient than F-NEWLEMMA.

**Search for counter-examples**

In Algorithm 3.1, if the lemmas do not imply stabilization then the counterexample search procedure, CEXSEARCH, is called to search for a counterexample, or exhaustively show that no counter-example exists. The procedure uses the bounds \( V_{\text{min}} \) and \( V_{\text{max}} \) computed earlier to limit the search space that is exhaustively explored.

CEXSEARCH is designed to find one of two types of counter-examples: multiple trivial fixpoints and non-trivial cycles. The searches for both types of counterexamples are done in parallel, and whichever is found first is returned. In the case of multiple fixpoints, CEXSEARCH encodes the problem of existence of at least two fixpoints as an instance of a satisfiability problem. A decision procedure is used to search for the existence of two different states: \( (v_1, \ldots, v_n) \) and \( (w_1, \ldots, w_n) \) such that each of them is a fixpoint:

\[
\forall i \in \{1 \ldots n\}. \left( v'_i = v_i \land w'_i = w_i \right) \land \exists i \in \{1 \ldots n\}. (v_i \neq w_i),
\]

where the system’s next states, \( (v'_i) \) and \( (w'_i) \), are determined by (3.2). We can ignore reachability here because the set of initial states is equal to the set of all possible state configurations. Note also that, for efficiency, we conjoin the system with extra constraints using \( V_{\text{min}} \) and \( V_{\text{max}} \):

\[
\forall v \in \{v_1, \ldots, v_n, w_1, \ldots, w_n\} \cdot V_{\text{min}}(v) \leq v \leq V_{\text{max}}(v).
\]

Experimentally we found that the information from the proved lemmas leads to tremendous speedups when searching for multiple fixpoints. Satisfiability
of the query proves the existence of at least two different fixpoints. If it is unsatisfiable, the system is cyclic or terminating; in the next phase we search for a non-trivial cyclic counter-example.

To find a non-trivial cycle we use bounded model-checking [28] together with the encoding of liveness to safety found in [29]. Namely, we check for the existence of cycles of length \( k = 2, 3, \ldots \), until a cycle is found or the system diameter reached, i.e. when \( k \) reaches the state space size, \( \prod_{v \in V} |V_{\text{max}}(v) - V_{\text{min}}(v)| \). The search for a cycle of length \( k \) is encoded as a formula satisfiability problem. A decision procedure is used to search for \( k \) states \( v_1, v_2, \ldots, v_k \) that form a cycle and are pairwise different:

\[
\forall j = 1 \ldots k . (v_j)' = v_j \land \forall j \neq l . v_j \neq v_l.
\]

For efficiency, as we unroll the system \( Q \) we conjoin it with constraints on the values of the variables that come from the proved lemmas

\[
\forall j = 1 \ldots k, i = 1 \ldots n . \ V_{\text{min}}(v_i^j) \leq v \leq V_{\text{max}}(v_i^j),
\]

as was done in the search for fixpoints. Again we find that the information from the proved lemmas leads to tremendous speedups when searching for non-trivial cycles. Termination of the unrolling uses a naive diameter check [28], leading to a sound and complete technique. Fortunately, we know only of toy examples where a search to the system’s diameter is necessary.

## 3.6 Formal proofs

This section gives formal proofs for the algorithms in Section 3.5.

**Theorem 3.1.** Procedure \textsc{GenLemmas}, displayed in Algorithm 3.2, is sound and complete.

**Proof.** A proof found by our procedure, \( \mathcal{P} \), consists of a list of local lemmas. The discovery order of the lemmas creates a partial order between them. Fix a complete linear order on the lemmas that linearizes this partial order. Let \( L_1, L_2, \ldots \) be this linear order, where \( L_i \) is \( (FG(p_i) \Rightarrow FG(q_i)) \), and \( q_i \) is a predicate on variable \( v_i \). We show by induction that for every \( i \), \( FG(q_i) \) holds.

**Base.** The first lemma to be discovered relies on nothing and is proved by model checking, or by Theorem 3.2 in case the optimized procedure is used.

**Induction step.** We prove that \( FG(q_i) \) holds. By the definition of the linear order, every conjunct in \( p_i \) appears as \( q_j \) for some \( j < i \). Thus, \( FG(p_i) \) holds by the assumption of the induction step. By soundness of model checking or Theorem 3.2, the call to \textsc{ProveLemma}(\( L_i, Q|v_i \)) establishes \( FG(p_i) \Rightarrow FG(q_i) \). We conclude that \( FG(q_i) \) holds.

Soundness follows, as we only accept a proof if, for each variable \( v_i \), a statement \( FG(v_i = k_i) \) for some constant \( k_i \) is a consequent of one of the lemmas. The state \( (k_1, \ldots, k_n) \) corresponds to the unique fixed-point in the definition of stabilization, see Section 3.4.

Completeness results from the trichotomy discussed in Section 3.5, over which all cases are covered by our procedure. Note that completeness depends on the finiteness of the networks studied and the language \( L \).
Table 3.1: Biological examples tested. $N+1$ indicates the granularity of the network; #variables and #edges represent the number of variables (components) and the number of interactions between variables (edges), respectively, in the model. The skin models Skin2D and Skin3D contain bugs that were detected for the first time by our tool. The repaired versions are suffixed with FXD.

<table>
<thead>
<tr>
<th>Model</th>
<th>N+1</th>
<th>#vars</th>
<th>#edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSKINFxD</td>
<td>4</td>
<td>25</td>
<td>45</td>
</tr>
<tr>
<td>SKINFxD</td>
<td>4</td>
<td>60</td>
<td>90</td>
</tr>
<tr>
<td>ESKIN6FXD</td>
<td>4</td>
<td>72</td>
<td>108</td>
</tr>
<tr>
<td>ESKIN7FXD</td>
<td>4</td>
<td>84</td>
<td>126</td>
</tr>
<tr>
<td>ESKIN8FXD</td>
<td>4</td>
<td>96</td>
<td>144</td>
</tr>
<tr>
<td>SKIN2DFXD</td>
<td>4</td>
<td>300</td>
<td>530</td>
</tr>
<tr>
<td>SKIN2D</td>
<td>4</td>
<td>300</td>
<td>530</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>N+1</th>
<th>#vars</th>
<th>#edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIN3DFxD</td>
<td>4</td>
<td>1200</td>
<td>2420</td>
</tr>
<tr>
<td>SKIN3D</td>
<td>4</td>
<td>1200</td>
<td>2420</td>
</tr>
<tr>
<td>DIABETES8D</td>
<td>3</td>
<td>75</td>
<td>148</td>
</tr>
<tr>
<td>DIABETES15W</td>
<td>3</td>
<td>75</td>
<td>148</td>
</tr>
<tr>
<td>VPC4</td>
<td>3</td>
<td>48</td>
<td>92</td>
</tr>
<tr>
<td>VPC6</td>
<td>3</td>
<td>72</td>
<td>138</td>
</tr>
<tr>
<td>P.R.(ECTOEVE)</td>
<td>4</td>
<td>7</td>
<td>23</td>
</tr>
</tbody>
</table>

Theorem 3.2. Consider variable $v$ and its inputs $w_1, w_2, \ldots, w_m$. We will prove that for any $S \subseteq [0, N]^n$

$$FG((w_1, w_2, \ldots, w_m) \in S) \Rightarrow FG(v \in [\min(T), \max(T)])$$

where $T = T_v(S)$. In other words, if values of inputs of a variable $v$ stay within $S$ at times $[t_0, \infty)$, the value of $v$ stays in $[\min(T_v(S)), \max(T_v(S))]$ at times $[t_0 + N, \infty)$.

Proof. Indeed, if $FG((w_1, \ldots, w_m) \in S)$, then there exists a time $t_0$ such that from that time onwards we have $(w_1, \ldots, w_m) \in S$, and so $T_v(w_1, \ldots, w_m) \in T$ and, in particular, $T_v(w_1, \ldots, w_m) \in [\min(T), \max(T)]$. Since $v$ changes at most by 1 per step according to the equation:

$$v'_i = \begin{cases} 
  v_i + 1 & v_i < T_i(v), \\
  v_i & v_i = T_i(v), \\
  v_i - 1 & v_i > T_i(v);
\end{cases} \quad (3.4)$$

no more than $N$ steps after $t_0$ the value of $v$ will reach, and never leave, $[\min(T), \max(T)]$. It follows that $FG(v \in [\min(T), \max(T)])$.

3.7 Experimental results

We have implemented Algorithm 3.1 in a tool called BIOCHECK, using Cadence SMV [27] as the implementation of PROVELEMA and Z3 [30] as the decision procedure. The NEWLEMMA procedure is easily parallelized: the local lemmas are proved in batches rather than one-by-one. All experiments were performed on a PC equipped with 4GB memory and a quad-core Intel processor with hyper-threading.
3.7. Experimental results

Biological systems tested

Information about the examples used during our experimental evaluation can be found in Table 3.1. These models are variations on four base systems: skin, diabetes, vulval development, and pair-rule genes.

The mammalian epidermis model [10], SkinFxD, consists of 5 cells, each containing 12 variables. We tested a simplified version, SSkinFxD, where only 5 variables per cell directly relevant to stabilization were considered (Figure 3.1). We also built elongated variants of this model: ones that consist of more than 5 cells, ESKIn6-8FxD, and ones that emulate multidimensional skin tissue. Skin2DFxD contains $4 \times 5$ cells (240 variables) and represents skin cross-section. Skin3DFxD consists of a $4 \times 5 \times 5 (=100)$ 3-D mesh of cells (1200 variables). Note that, using our tool, we are the first to find a bug in the skin model from [10] (Section 3.3).

The model of several molecular pathways operating in type-2 diabetes and chronic obesity [13], Diabetes, exists in two variants: 8 days and 15 weeks after mice started being fed a fatty diet.

The model of vulval precursor cells (VPCs) [14] represents the process of cell fate determination during the formation of vulva, an egg-laying organ, in the *C. elegans* worm (this process was discussed in Chapter 2). The VPC4 model includes 4 cells; in nature, there are 6 VPCs, but the model was reduced by its author to 4 cells to make analysis by other tools tractable. Our tool easily handles the extended model, VPC6, which includes 6 cells.

We also tested PairRule, a regulatory network of genes operating during segmentation in the *Drosophila* embryo [15], and a mutant of this network, PairRuleEctoEve, with ectopic (in the wrong place) expression of the even-skipped gene. In [15], Sanchez *et al* report the former model non-stabilizing and the latter stabilizing, which is confirmed by our results. As the pair-rule model is very small, the time to analyze it is negligible and is not included in the performance comparison.

Comparison against other tools

The comparison between our tool and existing tools is presented in Table 3.2. In this table we have compared the following tools:

- **BC** is our tool BioCHECK implementing PROVE (Algorithm 3.1) using NEWLEMMAS (Algorithm 3.3).
- **FBC** is BioCHECK with domain-specific optimization, *i.e.* using F-NEWLEMMAS (Algorithm 3.4) instead of NEWLEMMAS.
- **NAIVE** is an implementation of bounded model checking using a diameter check as the termination condition, *i.e.* NAIVE($Q$) = CEXSEARCH(∅, ∅, $Q$).
- **TRM$^\alpha$** is the application of TERMINATOR [21] to solve a slightly different problem than stabilization (as stabilization itself is not encodable using LTL). For all the models that do stabilize, we test if the provided fixpoint is eventually reached. For those that do not guarantee stabilization we look for a non-trivial cycle. We use the symbol $\alpha$ to remind the reader that this application is not solving quite the same problem as stabilization.
- **SMV$^\alpha_1$** and **SMV$^\alpha_2$** apply Cadence SMV and NuSMV [31], respectively, to the same problem as is used in TRM$^\alpha$. 
Table 3.2: Comparison of our approach with other tools. ‘P’ means that the model was proved stabilizing; ‘D’ means that stabilization was disproved. BC is found in Algorithm 3.1 in Section 3.5. FBC is the domain-specific version of BC using F-NewLemmas instead of NewLemmas. Runtimes are given in seconds. T indicates a timeout, where the threshold was set to 20 minutes. M represents an out-of-memory exception. The memory threshold was set to 4GB. V indicates tool failure after reporting too many variables or other kind of resources.

<table>
<thead>
<tr>
<th>Model</th>
<th>Result</th>
<th>BC</th>
<th>FBC</th>
<th>NAIVE</th>
<th>TRM^α</th>
<th>SMV^α_1</th>
<th>SMV^α_2</th>
<th>QNB</th>
<th>SPN^α</th>
<th>VIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSkinFxD</td>
<td>P</td>
<td>3.8</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>SkinFxD</td>
<td>P</td>
<td>9.0</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>ESkin6FxD</td>
<td>P</td>
<td>10.6</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>ESkin7FxD</td>
<td>P</td>
<td>12.9</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>ESkin8FxD</td>
<td>D</td>
<td>12.3</td>
<td>1.0</td>
<td>2.1</td>
<td>T</td>
<td>M</td>
<td>M</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Skin2DFxD</td>
<td>P</td>
<td>50.3</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Skin2D</td>
<td>D</td>
<td>56.5</td>
<td>13.1</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>T</td>
</tr>
<tr>
<td>Skin3DFxD</td>
<td>P</td>
<td>257.3</td>
<td>0.1</td>
<td>T</td>
<td>T</td>
<td>V</td>
<td>M</td>
<td>V</td>
<td>M</td>
<td>T</td>
</tr>
<tr>
<td>Skin3D</td>
<td>D</td>
<td>396.8</td>
<td>182.8</td>
<td>T</td>
<td>T</td>
<td>V</td>
<td>M</td>
<td>V</td>
<td>M</td>
<td>T</td>
</tr>
<tr>
<td>Diab.8D.</td>
<td>P</td>
<td>4.9</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Diab.15w.</td>
<td>P</td>
<td>5.2</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>VPC4</td>
<td>P</td>
<td>4.6</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>VPC6</td>
<td>P</td>
<td>7.0</td>
<td>0.0</td>
<td>T</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>M</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

- QNB is a tool from [10] that computes infinitely-often visited states in a network. For the comparison in Table 3.2, we could only use the tool that treats a system as a whole, rather than the version using the system’s hierarchical structure to accelerate the whole-system reachable states computation. This acceleration-based technique has not been implemented. When applied manually to the example Skin, on similar hardware, the acceleration-based technique took 21 minutes (see [10]). With some help by the author of the tool, we have established that the acceleration-based technique still would not be able to handle our larger examples.

- SPN^α is the application of Spin [32] on the same formulas as in TRM^α.

- VIS is used in our experiments to symbolically compute the model’s reachable state spaces, from which we look for a stable state.

Note that all previously known approaches fail to scale to the larger examples. For example, in the column TRM^α the encoding creates a program that, in essence, forces the liveness prover to find termination arguments for each possible path through the loop, which is a very large set (e.g. SkinFxD contains 3^{60} such paths). For this reason, TERMINATOR times out. In the case of SMV^α, the SkinFxD has 4^{60} reachable states, which exceeds the typical limits of symbolic model checking tools. Note that unlike Naive, our implementation of CEXSEARCH with range restrictions does scale. This shows how the range restrictions that come from the lemmas help reduce the state
Table 3.3: Experimental details of application of our tool to the examples. Proof size is the number of lemmas in the proof, if the stabilization was proved. Otherwise counter-example size is given: cycle length and number of variables involved in the cycle. All times are given in seconds. F-GENLEMNAS is the domain-specific GENLEMNAS procedure that uses the optimized generation of new lemmas F-NEWLEMNAS instead of NEWLEMNAS. CEX is short for counter-example.

<table>
<thead>
<tr>
<th>Model</th>
<th>Gen-Lemmas</th>
<th>F-Gen-Lemmas</th>
<th>Proof size</th>
<th>CEX Search</th>
<th>CEX Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKINFXD</td>
<td>9.0</td>
<td>0.0</td>
<td>177</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ESKIN6FXD</td>
<td>10.6</td>
<td>0.0</td>
<td>212</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ESKIN7FXD</td>
<td>12.9</td>
<td>0.0</td>
<td>251</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ESKIN8FXD</td>
<td>11.3</td>
<td>0.0</td>
<td>2</td>
<td>1.0</td>
<td>74</td>
</tr>
<tr>
<td>SKIN2D</td>
<td>43.4</td>
<td>0.0</td>
<td>2</td>
<td>13.1</td>
<td>215</td>
</tr>
<tr>
<td>SKIN2DFXD</td>
<td>50.3</td>
<td>0.0</td>
<td>926</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SKIN3D</td>
<td>14.1</td>
<td>0.1</td>
<td>2</td>
<td>182.7</td>
<td>860</td>
</tr>
<tr>
<td>SKIN3DFXD</td>
<td>257.3</td>
<td>0.1</td>
<td>3896</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VPC4</td>
<td>4.6</td>
<td>0.0</td>
<td>75</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VPC6</td>
<td>7.0</td>
<td>0.0</td>
<td>107</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DIABETES8DAYS</td>
<td>4.9</td>
<td>0.0</td>
<td>132</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DIABETES15WEEKS</td>
<td>5.2</td>
<td>0.0</td>
<td>132</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PAIRRULE</td>
<td>4.8</td>
<td>0.0</td>
<td>2</td>
<td>1.6</td>
<td>4</td>
</tr>
<tr>
<td>PAIRRULE(ECTOeve)</td>
<td>1.7</td>
<td>0.0</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Note that in the case of the non-stabilizing ESKIN8FXD algorithm, our lemma generation procedure performs worse than the naive method. This demonstrates (as mentioned in Section 3.1) that our lemma generation procedure could in some cases hinder rather than help performance.

Details of experiments
Table 3.3 contains more statistics about the results of BioCHECK during the experimental evaluation. The optimized lemma generation procedure performs an order of magnitude faster than the one that uses a model checker. The size of the counter-examples found corresponds to the number of variables in the network that haven’t been fixed by the proof procedure (not shown); meaning that the proof procedure comes close to a counter-example.

Scaling-up BioCHECK to millions of cells
In Table 3.4 we check how our proof procedure scales to larger examples. We run them on models containing up to $10^4$ cells (with or without bug) until we ran out of memory. The NewLemmas-based implementation does not time out on exactly one of these examples. In contrast, the F-NewLemmas-based implementation successfully verifies all but the $200 \times 500$ mesh model.
Table 3.4: Performance of our tool FBC on scaled-up variants of the Skin3DFxD model.
All times are given in seconds. M represents an out-of-memory exception.

<table>
<thead>
<tr>
<th>Mesh (#cells)</th>
<th>#Vars (N+1=4)</th>
<th>F-Gen-Lemmas</th>
</tr>
</thead>
<tbody>
<tr>
<td>10×10×5</td>
<td>6.0 · 10³</td>
<td>0.8</td>
</tr>
<tr>
<td>10×20×5</td>
<td>1.2 · 10⁴</td>
<td>1.6</td>
</tr>
<tr>
<td>20×20×5</td>
<td>2.4 · 10⁴</td>
<td>3.6</td>
</tr>
<tr>
<td>10×50×5</td>
<td>3.0 · 10⁴</td>
<td>4.5</td>
</tr>
<tr>
<td>20×50×5</td>
<td>6.0 · 10⁴</td>
<td>9.8</td>
</tr>
<tr>
<td>50×50×5</td>
<td>1.5 · 10⁵</td>
<td>25.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mesh (#cells)</th>
<th>#Vars (N+1=4)</th>
<th>F-Gen-Lemmas</th>
</tr>
</thead>
<tbody>
<tr>
<td>75×75×5</td>
<td>3.4 · 10⁵</td>
<td>57.4</td>
</tr>
<tr>
<td>100×100×5</td>
<td>6.0 · 10⁵</td>
<td>103.8</td>
</tr>
<tr>
<td>100×200×5</td>
<td>1.2 · 10⁶</td>
<td>208.5</td>
</tr>
<tr>
<td>200×200×5</td>
<td>2.4 · 10⁶</td>
<td>423.0</td>
</tr>
<tr>
<td>100×500×5</td>
<td>3.0 · 10⁶</td>
<td>544.3</td>
</tr>
<tr>
<td>200×500×5</td>
<td>6.0 · 10⁶</td>
<td>M</td>
</tr>
</tbody>
</table>

3.8 Summary and conclusions

In this chapter we have addressed the open problem of scalable stabilization proving with a new sound and complete modular proof procedure. Our procedure takes advantage of the fact that, in practice, we can limit the set of possible modular proofs from which we search to those where the local lemmas are of a very restricted form. This leads to tremendous speedups, both for proving as well as disproving stabilization. It seems that it is the inherent robustness of biological systems that makes our technique work so well—evolutionary developed systems remain naturally stable in the presence of timing and concentration variations.

Using BioCheck, we were able to prove stabilization for a 3-D mesh of 200×500×5 mammalian skin cells. The state space of this model contains 2⁶⁴mln reachable states, which is by far the largest state space that we approach in this dissertation. Verification based on abstract interpretation coupled with modular reasoning is successful because it does not explicitly enumerate states or paths of systems; instead, it over-approximates the execution of a system using data structures of linear size with respect to the number of concurrent components. The greatest limitation of this technique is the need for tailoring to classes of properties and systems, i.e. abstract interpretation must only extract information relevant to the property at hand, and different kinds of properties require different kinds of information. In the future, it would be interesting to adapt our technique to prove additional liveness properties beyond stabilization, as well as exploit the circular proof rules e.g. [7, 8] for the purpose of proving stabilization in very large but finite systems.

3.9 Bibliography


Distributed processing of large graphs

4.1 Introduction

Model checking [1] is a technique to automatically check properties of programs (models) by systematically examining their state spaces (see Chapter 1). This technique is easy to use and widely applicable, but it is also limited by the amount of memory available to store the state spaces, which tend to be large due to state explosion. We mention two important approaches to mitigate this problem. First, state space size can be reduced; a simplest such technique is on-the-fly model checking, in which state space exploration and checking run concurrently, allowing to find bugs in partially explored models. Second, more memory can be obtained by splitting a state space between memories of multiple machines—this approach is called distributed model checking. Importantly, such a distributed environment requires that the verification and state space reduction algorithms are parallelized, which we address in this chapter.

Model checking is a graph algorithm: it operates on graphs in which states of a system are vertices, and transitions between the states are edges. Such graphs have two important properties: they are large and sparse (the number of edges per state is small compared to the total number of states); we observe that real-world graphs, i.e. graphs connecting people, places, etc., also exhibit these properties. For example, as of 2012: Facebook connects close to a billion users [2]; OpenStreetMap reports 1.5 billion of geographic locations [3]; and Google indexes a trillion unique web URLs [4]. These real-world graphs are sparse, as the number of links in a web page or the number of person’s friends are small compared to the size of the network.

In this chapter, we generalize the problem of implementing distributed model checking to the problem of implementing a general-purpose framework to write distributed algorithms that operate on large, sparse, partitioned
Distributed processing of large graphs

Following this idea, we design a graph processing model that is amenable to parallelization, and we realize it in a novel framework called HipG. Next, we use the framework to create and evaluate SpinJadi, our distributed enumerative on-the-fly model checker.

Writing distributed graph algorithms is challenging. First, it requires that a large unstructured graph is partitioned into chunks of balanced size in such a way that relatively few edges span different chunks, as each such edge incurs communication overhead; creating such a partitioning may be hard [5, 6]. Second, the amount of computation per graph node and edge is generally small, and the computations are unstructured, as they are typically driven by a node-edge relation. Consequently, the communication to computation ratio is often high, resulting in an immense communication overhead. In addition, graph computation tends to exhibit poor locality due to the graphs’ irregular structure [7]. Finally, on a distributed machine, obtaining good load balancing is hard, as in general work cannot be migrated (part of the graph would have to be migrated and all workers informed).

While for sequential graph algorithms a few graph libraries exist, notably the Boost Graph Library [8], for parallel graph algorithms no standards have been established. The current state-of-the-art amongst users wanting to implement distributed graph algorithms is to either use the generic C++ Parallel Boost Graph Library (PBGL) [9, 10] or, most often, create ad-hoc implementations, in which the main challenge is to implement communication. Not only does the ad-hoc coding effort have to be repeated for each new algorithm, but it also obscures the original elegant graph algorithm. A programmer spends considerable time tuning the communication, which is prone to errors. While it may result in a highly-optimized problem-tailored implementation, the code can only be maintained or modified with substantial effort.

In this chapter we propose HipG, a distributed framework aimed at implementing Hierarchical Parallel Graph algorithms that operate on large-scale graphs. Graphs can be read from disk, synthesized in memory, or created on-the-fly during execution of an algorithm. They can be pre-partitioned by the user or partitioned automatically by the framework. The key idea in HipG is to expose each graph node as an object with customizable data and methods, and deliver a unified interface to executing methods on local and non-local graph nodes. This way a graph node can seamlessly execute methods on any other graph node. In particular, nodes may execute methods on their neighbors, which execute methods on the neighbors’ neighbors, etc., which results in a fine-grained, recursive, structure-driven computation. These computations are parallelized automatically by HipG, which handles details of an execution on a parallel machine, and thus allows the user to focus on the algorithm at hand. Separation of the graph algorithm from communication details enables the algorithm to be expressed elegantly and easy to modify.

In HipG, the user controls the fine-grained graph computations by way of special objects called synchronizers. Synchronizers may spawn, await, and stop graph computations, as well as gather global results of such computations. Importantly, they can also spawn new synchronizers, which execute independently in parallel. This feature allows to create divide-and-conquer (i.e. hierarchical) graph algorithms: sub-synchronizers can solve sub-problems on
4.1. Introduction

sub-graphs. While this will be especially useful in Chapter 5, in the current chapter we give an example using decomposition into strongly connected components (see Section 4.4). Another graph processing model that can be expressed in HipG is the bulk synchronous parallel (BSP) [11] model, which alternates computation with communication phases; we give an example of a BSP program (breadth-first search) written in HipG in Section 4.3.

Although the user must be aware that a HipG program runs in a distributed environment, the code is high-level: explicit communication is not exposed by the API, nor are the algorithms tied to graph representations. Parallel composition is done in a way that does not allow race conditions, so that no locks or thread synchronization code have to be implemented by the user. These facts, coupled with the use of an object-oriented language, makes for an easy-to-use, but expressive, language to code parallel graph algorithms.

In this chapter we also introduce SpinJadi (Section 4.5), our distributed enumerative on-the-fly model checker, and a reimplementation of the model checker SPIN [12]. The input to SpinJadi is a multithreaded program in the Promela language [12]. The model checker starts with an empty graph, explores the system’s initial state, its successors, the successors of the successors, and so on, until a bug is found in the partially-explored state space, or the entire state space is exhausted. It is the input file that determines the next neighbor function that HipG uses to on-the-fly compute successors of states.

SpinJadi partitions the state space between the workers using hashing; each new state is explored locally, or sent to the worker that owns it. Safety properties are checked on states during exploration, but the crucial and most challenging part of the tool is checking properties about the system’s infinite executions, for instance that a certain state is always eventually reached. Such properties are described in an input program by way of annotating certain ‘bad’ states as accepting; to determine that a property does not hold, we must find a cycle that contains an accepting state. We use an algorithm for distributed cycle detection by Brim et al [13], which is based on finding a vertex that is its own maximal predecessor and thus must be on a cycle.

We implemented HipG in Java. See Section 4.6 for a discussion of this choice, as well as other implementation details. Using HipG we processed on our cluster [14] graphs of size of the order of $10^{10}$, and obtained good performance (see Section 4.7). SpinJadi was tested on two mutual exclusion protocols from the BEEM repository [15], as well many biological models formalized as Regulatory Networks [16] (see Chapter 3) from the GINsim repository [17]. We report on the model of T-cell activation during an immune response, one of the largest models in the repository.

We find that programs written in HipG are in general short and elegant: a program for decomposing a graph into its strongly connected components in HipG is an order of magnitude shorter than the hand-optimized C/MPI version of this program [18] and three times shorter than the corresponding implementation in PBGL (Section 4.2).

The remainder of this chapter is organized as follows. Section 4.2 overviews related work. The model and usage of HipG are explained in Section 4.3, and the ability to create divide-and-conquer algorithms is
illustrated in Section 4.4. In Section 4.5 we present on-the-fly computations in HipG, and in particular our distributed model checker SpinJadi. The implementation of HipG is discussed in Section 4.6, followed by an evaluation in Section 4.7. Section 4.8 concludes this chapter.

This chapter has been published before, as:
which is an extended (with SpinJadi) version of the conference paper:

4.2 Related work

HipG is a distributed framework aimed at providing users with a way to code, with little effort, parallel algorithms that operate on partitioned graphs. An analysis of other platforms suitable for the execution of graph algorithms is provided in an inspiring paper by Lumsdaine et al [7] that, in fact, advocates using massively multithreaded shared-memory machines for this purpose. However, such machines are very expensive and software support is lacking [7]. The library in [19] realizes this concept on a Cray machine. Yet another interesting alternative would be to use partitioned global address space languages like UPC [20], X10 [21] or ZPL [22], but we are not aware of support for graph algorithms in these languages, except for the shared memory solution [23] based on X10 and Cilk.

A graph programming framework that most closely resembles HipG is the Signal/Collect [24] framework targeted at the Semantic Web community. In Signal/Collect graph computations are expressed in terms of signals sent along edges, which correspond to HipG’s execution of methods on graph nodes. An advantage of the Signal/Collect model is that the scheduling of signals allows for malleability: the model provides synchronized, asynchronous, and prioritized executions. However, similarly to Pregel, the user controls signals but not the global execution; HipG allows the global execution to be defined by the user (via synchronizers). In addition, Signal/Collect is only implemented for shared memory systems.

The prominent sequential Boost Graph Library (BGL) [8] gave rise to a parallelization that adopts a different approach to graph algorithms. Par-
allel BGL [9, 10] is a generic C++ library that implements distributed graph data structures and graph algorithms. The main focus is to reuse existing sequential algorithms, only applying them to distributed data structures, to obtain parallel algorithms. PBGL supports a rich set of parallel graph implementations and property maps. The system keeps information about ghost (remote) vertices, although that works well only if the number of edges spanning different processors is small. Parallel BGL offers a very general model, while both Pregel and HipG trade expressiveness (for example neither offers any form of remote read) for more predictable performance. ParGraph [25] is another parallelization of BGL, similar to PBGL, but less developed; it does not seem to be maintained. We are not aware of any work directly supporting the development of divide-and-conquer graph algorithms.

The Bulk Synchronous Parallel (BSP) model of computation [11] alternates work and communication phases. We know of two BSP-based libraries that support the development of distributed graph algorithms: CGMgraph and Pregel. CGMgraph [26] uses the unified communication API and parallel routines offered by CGMlib, which is conceptually close to MPI [27]. In Google’s Pregel [28] the graph program is a series of supersteps. In each superstep the Compute(messages) method, implemented by the user, is executed in parallel on all vertices. The system supports fault-tolerance consisting of heartbeats and checkpointing. Impressively, Pregel is reported to be able to handle billions of nodes and use hundreds of workers. Unfortunately, it is not available for download. Pregel is similar to HipG in two aspects: the vertex-centered programming and composing the parallel program automatically from user-provided simple sequential-like components. However, the repeated global synchronization phase in BSP, although suitable for many applications, is not always desirable. HipG is fundamentally different from BSP in this respect, as it uses asynchronous messages with computation synchronized on the user’s request. Notably, HipG can simulate the BSP model, which is shown in the breadth-first search implementation in Section 4.3.

What HipG does not currently support is combining the memory with external storage. In [29] some nodes created during enumerative model checking are stored on disk and accessed through Bloom filters to reduce the number of I/O operations. In [30] parts of the graph are stored on solid-state memory devices that are significantly faster than disks. Both solutions were designed for shared memory systems.

Besides general graph programming frameworks, tailored solutions to some parallel graph problems exist. In the formal methods community a number of distributed model checkers were developed to cope with the state explosion problem. The DiVinE LTL model checker [13, 18, 31, 32] can utilize both multi-cores and distributed memory. DiVinE is highly optimized for performance [33]. Another notable model checking tool, LTSmin [34], introduces a new high-level layer in which new algorithms and new interface languages can be plugged in. Both tools are implemented in MPI/C++.

To store graphs we used the SVC-II distributed graph format [35]. Graph formats are standardized only within selected communities. In case of large graphs, binary formats are typically preferable to text-based formats, as compression is not needed. See [35] for a comparison of a number of formats.
Distributed processing of large graphs

Figure 4.1: Reachability search that starts from p and leads to the gray colored nodes distributed at three machines.

used in the formal methods community. A popular text format is XML, which is used for example to store OpenStreetMap [36]. RDF [37] is used to represent semantic graphs in the form of triples (source, edge, target). Najork [38] describes how the web graph can be compactly stored in memory. By contrast, in bioinformatics, graphs are stored in many databases and integrating them is ongoing research [39].

4.3 Basic model and API

The input to a HipG program is a directed graph; HIPG partitions the graph into chunks of equal size. A chunk is a set of graph nodes and their outgoing edges; in other words, edges are co-located with their source nodes. The target node of an edge is called a neighbor or a successor. Undirected edges are modeled as two directed edges. Each node is an object containing user-defined arbitrary data and a unique identifier, for example by a pair (chunk, index). Chunks are given to workers who are responsible for processing nodes that belong to them.

Graphs are typically processed by following their structure, i.e. the node-edge relationship. For example, an algorithm may start processing at a pivot node, then process its neighbors, the neighbors’ neighbors, etc., until all reachable nodes have been processed—such an algorithm is called a reachability search, and is illustrated in Figure 4.1. In HipG this is realized by allowing the user to define custom methods on graph nodes, and providing a unified interface to execute methods on local and non-local nodes, as well as a seamless access to a node’s list of neighbors. This is explained in detail by the following example.

Example: Reachability search

Figure 4.2(a) displays the core of the reachability search implementation in HipG in Java. First, a node interface is defined, MyNode, telling HipG which methods can be executed on remote nodes. In general, methods listed in the interface can be executed on any graph node of which an identifier is known. MyLocalNode is the node implementation. Each node has a flag that denotes whether it has been visited. The visit() method visits an unvisited node and its neighbors. The parts underlined in the code are provided or required by HipG, the remaining parts are user-defined. There are several essential observations to be made about the code in Figure 4.2(a). First, no locks or other methods of
synchronization were needed; the exclusive access to the node is assured by the framework. Lack of synchronization makes the code look sequential and therefore easy to program. Nevertheless, the user must be aware that the code will execute in a parallel setting: the order in which methods execute cannot be predicted and relied upon in the algorithm; even on a single processor, HipG might reorder node method calls to prevent stack overflow. Second, the layout of the graph data structures is not exposed to the user; in fact, not only may the actual data structure vary in various graph implementations, but parts of it might not even be created yet (as in Section 4.5). Finally, the user did not need to provide different handling of local and non-local neighbors: access to all graph’s nodes is unified. All these facts make HipG node methods easy to read and high-level: the code reflects the algorithm behind it.

**Synchronizers**

The algorithm in Figure 4.2(a) is initiated at the pivot node and terminates when all reachable nodes have been processed. In HipG this is written as: pivot.visit() followed by a barrier(), see Figure 4.2(b). This code is, in fact, the simplest example of a synchronizer, a logical object that manages distributed computations. The three basic operations of a synchronizer are:

- It initiates distributed computations, which execute in parallel or in sequence. For example, the call to pivot.visit() starts a ‘wave’ of visit() method calls as in Figure 4.1.
- It awaits termination of all computations issued directly and indirectly; this is achieved using barrier(). A barrier blocks the synchronizer until all computations initiated by this synchronizer have completed. For example, the barrier after pivot.visit() blocks until all reached nodes have been visited and there are no visit()'s in transfer.
- It gathers results of distributed computations, e.g. a globally elected pivot, or a size of a set of nodes partitioned between workers (see the
Distributed processing of large graphs

4.3 Breadth-first search in HipG.

next example in this section).

One can imagine a synchronizer as an ‘agent’ that manages distributed graph computations on behalf of the user. For each logical synchronizer, one instance of it is delegated to each worker machine; these instances communicate, for example to determine termination of computations. Each logical synchronizer has a unique id, determined on spawn, and consistent across all workers (see Section 4.6).

Example: Breadth-first search

Figure 4.3 shows the breadth-first search (BFS) implemented in HipG. The aim of BFS is to compute—layer by layer—all nodes reachable from a pivot node. The major part of our BFS implementation is the BFS synchronizer, of which an instance executes on each worker. BFS maintains a queue Q of nodes in the current layer, partitioned between the workers. The worker that owns the pivot, inserts it into the queue (line 6). The run() method loops over the nodes in the current layer (line 12), and appends their unvisited neighbors to Q (not shown) in the method found (line 13), in this way building the new layer. Note that the newly discovered nodes may be added to the local queue or a queue on the worker that owns the neighbor. The barrier (line 14) blocks until the new layer is fully created, i.e. until found messages have been processed, and there are no messages in transit. Building layers terminates when an empty layer is reached (line 16); to this end, the global size of Q is computed by GlobalQsize using a special Reduce annotation provided by HipG. Without it,
the call to $GlobalQsize$ would be a regular method call returning the size of $Q$; with it, HipG automatically at compile time translates the call to $GlobalQsize$ into a global reduce operation, which blocks until each worker executes this method and the final result is obtained. Each call to a reduce operation combines a partial result with local data, and forwards the updated partial result to another worker. In our case, the result will be a sum of sizes of all $Q$s. We note that (i) each worker executes the reduce method exactly once; (ii) the execution blocks until the result of the reduction is obtained; (iii) the final result is consistent across all workers; and, importantly, (iv) the order of execution of reduce operations cannot be predicted and relied upon in the user’s code, so the operation must be commutative.

We note that synchronizers only use high-level communication routines such as barriers and reduce operations; no synchronization mechanisms are needed, even if there are multiple synchronizers per worker. Conceptually, the framework executes each $run()$ method in isolation and sequentially, with exclusive access to the synchronizer’s data structures, and independently of other synchronizers. Note that BFS alternates computation with global synchronization, which follows the bulk synchronous parallel (BSP) model [11].

**Lifting to parallel applications**

The two examples above, Visitor in Figure 4.2 and BFS in Figure 4.3, show that a user of HipG writes graph algorithms by defining two components: graph nodes with custom data and methods, and one or more synchronizers’ $run()$ methods. The former represent structure-driven computations on graph nodes, whereas the latter manage such graph computations. These two components constitute the whole HipG program.

Note that the user-defined code in the two examples looks sequential: it is the model that ties it into a parallel application. The runtime system automatically lifts a HipG program into a parallel application on a distributed machine. Importantly, at compile-time, it translates calls to methods on non-local nodes into asynchronous messages. Since messages are asynchronous, methods must not return values. Returning a value of a method can be realized by sending a message back to the source, although, typically, HipG’s dedicated mechanism of computing global results (reduction) is more efficient.

### 4.4 Divide-and-conquer graph algorithms

Divide-and-conquer graph algorithms divide computations on a graph into several sub-computations on sub-graphs. HipG enables creation of sub-algorithms by allowing synchronizers to spawn any number of sub-synchronizers. Therefore, a HipG algorithm is, in fact, a tree of executing synchronizers, and thus a hierarchy of distributed algorithms. Synchronizers can manage child synchronizers, for example wait for child termination. Unless explicitly synchronized, all synchronizers execute independently and in parallel. The user starts a graph algorithm by explicitly creating and spawning the root synchronizer. The system terminates when all synchronizers terminate. We illustrate divide-and-conquer graph algorithms in HipG with an example of decomposition into strongly-connected components.
Distributed processing of large graphs

**FB(V):**
- \( p = \text{pick a pivot from } V \)
- \( F = \text{FWD}(p) \)
- \( B = \text{BWD}(p) \)

Report \((F \cap B)\) as SCC

In parallel:
- \( \text{FB}(F \setminus B) \)
- \( \text{FB}(B \setminus F) \)
- \( \text{FB}(V \setminus (F \cup B)) \)

![Figure 4.4: Divide-and-conquer SCC-decomposition.](image)

**Strongly-connected components**

A strongly connected component (SCC) of a directed graph is a maximal set of nodes \( S \) such that there exists a path in \( S \) between any pair of nodes in \( S \). We briefly describe FB [40], a divide-and-conquer graph algorithm for computing SCCs, and sketch its implementation in HipG. The concept is explained in Figure 4.4. FB partitions the problem of finding SCCs of a set of nodes \( V \) into three sub-problems on three disjoint subsets of \( V \). First an arbitrary pivot node is selected from \( V \). Two sets \( F \) and \( B \) are computed as the sets of nodes that are, respectively, forward reachable and backward reachable (i.e. reachable in the transposed graph) from the pivot. The set \( F \cap B \) is an SCC. All SCCs remaining in \( V \) must be entirely contained either within \( F \setminus B \) or within \( B \setminus F \) or within the complement set \( V \setminus (F \cup B) \).

The crucial part of FB implementation is displayed in Figure 4.5. The synchronizer starts by selecting a global pivot from \( V \) (line 4) with the SelectPivot reduce operation (discussed earlier). The pivot owner initializes forward and backward reachability searches that create sets \( F \) and \( B \) in \( V \) (lines 6–9) by flagging the reached nodes and storing them in separate queues (not shown). After \( F \) and \( B \) are fully computed, three sub-synchronizers are spawned to solve three sub-problems on \( F \setminus B \), \( B \setminus F \) and \( V \setminus (F \cup B) \).

Note that FB uses the input graph as well as its transpose, which has to be provided by the user or can be computed by HipG (in fact, the transpose computation is a graph algorithm written in HipG); HipG contains routines that can access the transpose: \( \text{hasInNeighbor}(i) \) and \( \text{inNeighbor}(i) \), similar to \( \text{hasNeighbor}(i) \) and \( \text{neighbor}(i) \) in Figure 4.1. Most importantly, we observe that Figure 4.5 elegantly reflects the algorithm in Figure 4.4. A corresponding C/MPI application (see Section 4.7) has over 1700 lines of code that entirely obscures the algorithm, and the PBGL implementation has 341 lines, while the entire FB in HipG is only 113 lines.

**4.5 On-the-fly graph algorithms**

Encapsulating graph data structures and exposing only a high-level graph interface to the user, makes HipG highly malleable. Not only are algorithms not tied to particular graph representations, but also graphs can be created on-the-fly, i.e. during execution: a node is created on first access to it.
4.5. On-the-fly graph algorithms

```java
class FB extends Synchronizer {
    ...
    public void run() {
        MyNode pivot = SelectPivot(null);
        if (pivot == null) return;
        if (pivot.isLocal()) {
            pivot.fwd(this, flags);
            pivot.bwd(this, flags);
        }
        barrier();
        spawn(new FB(F \ B));
        spawn(new FB(B \ F));
        spawn(new FB(V \ (F ∪ B)));
    }
}
```

Figure 4.5: FB algorithm in HipG.

This allows overlapping graph creation with computation for performance, and is essential in cases when the algorithm only requires a part of the graph to execute, while the entire graph would not fit in the memory. To generate a graph on-the-fly, the user provides a definition of a next neighbor function prior to execution. Using this feature we implemented a distributed model checker, which otherwise might have taken months to develop from scratch. Distributed model checkers already exist, most notably DiViNe [31] (see Section 4.2), and are typically large projects; the high-level API of HipG allows to vastly speed up development and try new algorithms with little effort.

Distributed model checking

We implemented SPINJAD1, a distributed enumerative on-the-fly model checker based on SPINJA [41], a recent Java reimplementation of SPIN [12], a state-of-the-art sequential model checker. The input to our model checker is a Promela [12] file, which represents a multithreaded program augmented with assertions and an LTL (see Chapter 1) property to be checked. In enumerative on-the-fly model checking, the reachable states of a model are explored and checked concurrently; the initial state is explored first, then successors of the initial state, successors of the successors, etc.

Two algorithms play a major role in distributed enumerative LTL model checking: on-the-fly visitor and MAP; SPINJADI invokes one of them, depending on the options supplied by the user. The first algorithm, the on-the-fly visitor, computes reachable states and checks them; it is implemented similar to the code in Figure 4.2(a), only augmented with many checks on visited states: maximal depth, assertions, deadlocks. The states are generated on-the-fly from the input program using a next neighbor function we adapted from

1SPINJA rhymes with Ninja; SPINJADI rhymes with Jedi
Distributed processing of large graphs

Figure 4.6: Maximal accepting predecessors (MAP) algorithm in HipG. The map values are initialized to ⊥, which denotes a special value smaller than any other value: ⊥ < −∞ < id().

SpinJa. The visitor terminates when an error is found, or the state space is exhausted.

The second algorithm checks properties of infinite executions of the model (for example a property that a certain ‘stable’ state is eventually reached from any other state), which is more challenging. We used Maximal Accepting Predecessors (MAP), a distributed algorithm by Brim et al [13], which searches for accepting cycles. Namely, a user describes the illegal infinite executions in the input file by way of annotating certain ‘bad’ states as accepting. Existence of an accepting cycle, i.e. a cycle that contains an accepting state, proves that the property under consideration does not hold. MAP assumes that the graph nodes have unique totally-ordered identifiers. It relies upon the observation that an accepting cycle exists if and only if there exists a graph node with itself as its maximal (with respect to identifiers) accepting predecessor. Therefore, in each iteration, MAP computes the maximal accepting predecessor of each node. If one of the accepting nodes is its own maximal accepting predecessor, an accepting cycle is reported. Otherwise, nodes which cannot be on an accepting cycle are discarded and the next iteration is started. In [13] this algorithm is described in detail and its correctness argues; next, we briefly discuss how it was implemented in HipG.
4.6 Implementation

MAP implementation in HipG

Figure 4.6(a) shows computation of the maximal accepting predecessor for graph nodes. Each graph node has a flag that says whether the state is accepting; this value is initialized (not shown) from the input program when the node is created. The identifier of the current maximal accepting predecessor is stored in the variable map, and forwarded to neighbors. If a node receives its own identifier, an accepting cycle is reported (lines 13–14). Otherwise, only map values greater than the current value are accepted (line 15) for propagation (lines 19–20); an accepting node propagates the maximum of its map and its identifier (lines 17–18). The computation terminates when all map values stabilize.

The MAP synchronizer, which describes the whole MAP algorithm, is displayed in Figure 4.6(b). In each iteration, it computes all map values (lines 7–9). If an accepting cycle was reported by one of the graph nodes (this is realized by way of asynchronous notifications, not shown), MAP terminates (lines 11–12). Otherwise, it discards the accepting nodes that cannot be on an accepting cycle (lines 15–17), i.e. when \( \text{map} < \text{id} \) (see [13]), and restarts the map computation (line 18). When no accepting cycle is found and the state space is exhausted and all accepting states discarded (line 21), no accepting cycle exists. The AcceptingNodesLeft() is implemented using HipG’s reduction mechanism described in Section 4.3.

4.6 Implementation

HipG is designed to execute in a distributed-memory (message-passing) environment. We chose to implement HipG in Java because of its portability and performance (due to the just-in-time compilation) as well as an excellent software support of the language, although Java required us to carefully ensure that memory is utilized efficiently. We used the Ibis [42] message-passing communication library and the Java 6 virtual machine implemented by Sun [43].

A HipG program is executed by a number of workers. Each worker stores a single chunk of the graph. Logically, HipG executes a set of synchronizers in parallel. In this section we describe the implementation of workers and synchronizers, and briefly mention the compile-time instrumentation, that provides the syntactic sugar of a seamless graph interface without any language extensions.

Graph storage on a worker

The input to a HipG program is a directed graph (or graphs). If the graph is not pre-partitioned by the user, HipG partitions it into chunks of equal size by uniformly hashing each node to an owner. Currently the number of edges spanning different chunks is not minimized. Each worker stores the entire chunk in memory. A chunk is a collection of graph nodes and their outgoing edges. Two chunk layouts are currently implemented: explicit and map. In the explicit layout all nodes are stored in an array and uniquely identified by a pair of integers (worker, index). Edges on a worker are stored as two global arrays, one for references to local nodes and one for identifiers.
of remote nodes. Although this structure is not elegant, it is transparent to the user and memory-efficient, as it minimizes the prohibitive per-object memory overhead of garbage-collected languages (16 bytes per-object in 64-bit HotSpot). The explicit layout is efficient but difficult to modify at run-time, which is in contrast with the map implementation based on a hash table. The user defines a key of this table and a hashing method. The map representation is used in on-the-fly graph algorithms. In the distributed model checker that we implemented, the key used was a byte array that represents a state in the checked program. In the on-the-fly algorithms the edges are not stored, but generated by the user's successor function. Last but not least, as most of the worker's memory is used to store the graph, we tuned the garbage collector to use a relatively small young generation size (5–10% of the heap size).

Communication
The HipG workers communicate intensively all-to-all. Messages representing execution of methods on remote nodes account for the bulk of the traffic between the workers. Workers only execute methods on graph nodes that they own (the 'owner-computes' rule). Each method call belongs to some synchronizer, which is supplied as the first argument of the method (compare with the example in Figure 4.6(a)(line 12)). A message consists of an identifier of a synchronizer it belongs to, an identifier of a graph, an identifier of the target node, and serialized arguments. Arguments are serialized automatically, and we strive to make the serialization efficient. On reception of a message, the target node is retrieved; if it belongs to a graph generated on-the-fly, the node might not exist yet, in which case it is created and stored. Further, the method to execute is decoded from the message, and the parameters are de-serialized and passed to the method. During the execution of the method, new method calls can be spawned. If such a method is called on a local node, it is executed right-away (until some depth), or stored on the synchronizer's queue (when that depth is exceeded) to be processed later. If the method is called on a remote node, it is buffered for sending. The messages are combined in non-blocking buffers and flushed repeatedly by the worker's sender thread. Asynchronous receiving is performed by a pool of threads provided by the Ibis communication library.

Synchronizer implementation
Each synchronizer is executed by all workers. Each synchronizer has a unique identifier, determined on spawn by the worker with rank 0, and broadcast to all workers. A synchronizer can spawn any number of sub-synchronizers,
so it also maintains information about its father and children. An example of the tree of synchronizers (synchronizer objects on workers) is shown in Figure 4.7, where numbers indicate synchronizer's identifier and dotted lines represent instances of the same synchronizer. The execution of a synchronizer, i.e. its run() method (which is executed by each worker), can be understood as alternating communication phases, when methods on nodes are executed, and synchronization phases, i.e. blocking because of a barrier or a reduction operation. Barriers are implemented with the token-based distributed termination detection algorithm by Safra [44]. When a barrier returns, it means that all method calls that belong to the synchronizer have been processed and there are no messages in transit. The reduce operation is also implemented by token traversal [45], and the result announced to all workers by the worker with rank 0. Notifications (used in Section 4.5) are implemented as acknowledged asynchronous messages.

Worker implementation

After reading the graph, the user's main() program typically initiates root synchronizers, waits for all synchronizers to terminate, and handles the computed results. The part of the runtime that executes synchronizers we refer to as a worker. A worker is a single thread that stores all local instances of synchronizers, and emulates execution of multiple independent synchronizers in parallel—by looping over a queue of active synchronizers. If a synchronizer is ready to progress (a blocking routine has just terminated), the worker executes the next ‘step’ (see next paragraph), of the run() method, or terminates the synchronizer, when run() has finished. A worker terminates when all synchronizers have terminated.

Program instrumentation

Before executing, HipG programs have to be instrumented. The Ibis rewriter [42] optimizes object serialization. The HipG rewriter translates remote method calls into messages, and breaks down the run() methods into ‘steps’ at each blocking routine. Thanks to this, a single worker thread can execute all synchronizers, without the need for many context switches, while the user perceives a synchronizer's run() method as a thread. The ‘stepping' of the run() method is accomplished by bytecode rewriting: each step of the run() method finishes with a checkpoint, and starts with reading such a checkpoint. Instrumentation is part of the provided HipG library, and needs to be called before execution. No special Java compiler is necessary.

4.7 Evaluation

In this section we report on the results of experiments conducted with HipG. The evaluation was carried out on the VU-cluster of the DAS-4 system [14]. The cluster consists of 74 dual quad-core 2.4 GHz Intel Xeon CPUs, with 24 GB of memory per compute node. The processors are interconnected with 32Gbps-capable 4xQDR InfiniBand. The time to initialize workers and input graphs was not included in the measurements.
All graphs were partitioned randomly—meaning that if a graph is partitioned in $p$ chunks, a graph node is assigned to a chunk with probability $1/p$. The portion of remote edges is thus $(p-1)/p$, which is very high (75–99% in used graphs) and realistic to model an unfavorable partitioning (many edges spanning different chunks). An advantage of this scheme is load-balancing: the numbers of edges stored at the workers are likely to be similar. We also note that in this setting, when computing a graph problem with twice as many workers, $2 \cdot p$, the amount of computation stays constant, but the volume of communication increases by a factor $1 + 1/(2(p-1))$. For $p \geq 4$, used in this evaluation, this factor is below 17%.

Message-based applications

We start with the evaluation of performance of applications that almost solely communicate (only one synchronizer spawned). VISITOR, which implements the reachability search (see Figure 4.2), was started at the root node of a large binary tree directed towards the leaves. BFS, a breadth-first search (see Figure 4.3), was started at a random node of a synthetic social network. Both graphs are stored explicitly in memory prior to execution. The results are presented in Table 4.1 (the first two applications) and Figure 4.8. We tested both applications on 2–64 processors, running two workers per compute node (two workers per node is a compromise: we want to test the framework using as many workers as possible, each handling a largest possible piece of memory).

To obtain more fair results, rather than keeping the problem size constant, we double the problem size when we double the number of workers. We note that this can only be done for very regular graphs and computation structure, and in this case we expect constant numbers for the first two applications in Table 4.1. Thanks to this we are able to test the graph applications on a wide range of workers; additionally, we avoid spurious improvements due to better cache behavior, keep the heap filled, but also avoid too many small messages that occur if the stored portion of a graph is small. Next, we argue the correctness of the speedup computation by doubling the problem size with doubling the number of workers. Let $T_p(s)$ denote execution time on $p$ workers on problem of size $s$. A graph problem is ‘regular’, if solving a doubled problem with the same number of workers takes twice as long, i.e. $T_p(s) = T_p(2s)/2$, for any $p$, $s$, which is the case of VISITOR/Bin-$n$ and BFS/LN-$n$, when $n$ is not very small. The speedup plotted is given by the formula $p \cdot T_{\min}(s_{\min})/T_p(p \cdot s_{\min})$, thus equal to the ‘traditional’ speedup formula. We compute speedup against the time $T_{\min}$ it takes the smallest set of workers to solve a given parallel problem (on two machines, each with two workers, $\min = 4$).

For VISITOR we used binary trees, Bin-$n$, of height $n = 27 \ldots 32$, i.e. up to $8.6 \cdot 10^9$ nodes and edges. The LN-$n$ graphs used for BFS are random directed graphs with degrees of nodes sampled from the log-normal distribution $\ln N(4.0, 1.3)$, aimed to resemble real-world social networks [28,46]. An LN-$n$ graph has $n \cdot 10^7$ nodes and expected $n \cdot 1.3 \cdot 10^9$ edges. We used LN-$n$ graphs for $n$ up to 64 and thus up to $6.4 \cdot 10^8$ nodes and $8.1 \cdot 10^9$ edges. In both experiments, all edges of the input graphs were visited. Both applications
4.7. Evaluation

Table 4.1: Performance of VISITOR, BFS and SPINJADI. #W means the number of workers; halved #W is the number of physical machines used. Time is given in seconds, and memory is given per worker in GB; M means out-of-memory. The speedup for all applications is shown and compared in Figure 4.8.

achieved at least 60% efficiency on 128 workers, which is satisfactory for applications with little computation, $O(n)$, compared to $O(n)$ communication. The efficiency achieved by BFS on LN-$n$ graphs reaches almost 80%, as the input is more randomized, and has a small diameter compared to a binary tree, which reduces the number of barriers performed.
On-the-fly applications

We performed two kinds of evaluations of the distributed model checker (see Section 4.5): assertion/deadlock checking SPINJADI-E, which enumerates the entire state space (ignores errors), and SPINJADI-A, which searches for accepting cycles. For both applications, we set unlimited exploration depth. We used the following models:

- Two examples from the BEEM repository of model checking benchmarks [15]: Peterson’s mutual exclusion protocol for 5 processes (PETERSON.7) and Anderson’s mutual exclusion protocol for 6 processes (ANDERSON.6). For the Anderson’s protocol we checked a liveness property that says that a process is in the critical section infinitely many times.
- A biological example, TCRsig29, which models signaling during T-cell activation. In this process, T-lymphocytes (also called T-cells) use a special receptor, TCR (T-cell Receptor), to detect foreign antigens, which leads to an immune response. This model was originally created by Klamt et al [47], and written as a Boolean Regulatory Network [16] by Naldi et al [48]. We obtained that model from the GINsim repository [17], and translated it to a Promela model with asynchronous updates using a methodology similar to the one proposed by Bošnački et al [49]. The original model had 40 components, of which we removed 11 without changing any of the semantics of the model\(^2\), in order to make the model more tractable. We could not repeat the steady state analysis of [48], i.e. when checking a given steady state, SPINJADI finds additional accepting cycles; this is likely due to the fact that the MAP algorithm does not take fairness into account. Instead, we used SPINJADI-A to check a simple safety property of TCRsig29.

We tested the SPINJADI-E and SPINJADI-A applications on 4–128 workers, as presented in Table 4.1 and Figure 4.8. As expected, SPINJADI-E scales similarly to VISITOR; the major difference between VISITOR and SPINJADI-E is that VISITOR allocates memory prior to execution, which is not timed, while SPINJADI-E allocates almost all memory during execution. SPINJADI-E verification of PETERSON.7 algorithm generates 142 million states, and 616 million transitions (on all workers combined); on TCRsig29, the state space reaches 83 million states and 1 billion transitions. The SPINJADI-A application also scales well; note that it is tested on properties that are expected to hold, and indeed in neither case it finds a bug. On the ANDERSON.6 model SPINJADI-A performs exceptionally well; it is not entirely clear why—it likely is due to particularly visible improved caching on more processors (ANDERSON.6 contains more atomic and d_steps than the other examples). Each run of the ANDERSON.6 generates about 10 million states and 1 billion of transitions; on TCRsig29, the state space SPINJADI-A contains 166 million states and 1 billion transitions.

\(^2\)We removed IP3, Ca, Calcin, SEK, JNK, Jun, Fos, Rsk, CREB, Raf, MEK. Neither of these components is a part of any feedback loop; in fact, all of them are cascade elements near ‘outputs’ of the model, and removing them only shortens cascades such as PLCg_a → IP4 → Ca → Caldin → NFAT (NFAT has no outputs) to PLCg_a → NFAT.
Table 4.2: Performance of OBFR-MP.

<table>
<thead>
<tr>
<th>$p$</th>
<th>Myri</th>
<th>HipG</th>
<th>Eth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MX</td>
<td>OM</td>
<td>P4</td>
</tr>
<tr>
<td>L487487T5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>36.6</td>
<td>141.4</td>
<td>41.1</td>
</tr>
<tr>
<td>8</td>
<td>26.6</td>
<td>81.6</td>
<td>22.1</td>
</tr>
<tr>
<td>16</td>
<td>96.5</td>
<td>60.5</td>
<td>48.4</td>
</tr>
<tr>
<td>32</td>
<td>40.0</td>
<td>57.3</td>
<td>39.1</td>
</tr>
<tr>
<td>64</td>
<td>24.1</td>
<td>46.7</td>
<td>24.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>234.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>41.8</td>
</tr>
<tr>
<td>L10L10T16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>69</td>
<td>255</td>
<td>148</td>
</tr>
<tr>
<td>8</td>
<td>73</td>
<td>280</td>
<td>226</td>
</tr>
<tr>
<td>16</td>
<td>89</td>
<td>376</td>
<td>315</td>
</tr>
<tr>
<td>32</td>
<td>136</td>
<td>661</td>
<td>485</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>646</td>
<td>277</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1659</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>461</td>
</tr>
<tr>
<td>L60L60T11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>45.1</td>
<td>152.9</td>
<td>47.3</td>
</tr>
<tr>
<td>8</td>
<td>34.5</td>
<td>99.8</td>
<td>46.8</td>
</tr>
<tr>
<td>16</td>
<td>37.1</td>
<td>128.6</td>
<td>60.4</td>
</tr>
<tr>
<td>32</td>
<td>30.1</td>
<td>82.0</td>
<td>57.4</td>
</tr>
<tr>
<td>64</td>
<td>32.0</td>
<td>108.8</td>
<td>66.1</td>
</tr>
</tbody>
</table>

Synchronizer-based applications

To evaluate the performance of hierarchical graph algorithms written in HipG, we ran the OBFR-MP algorithm [18] that decomposes a graph into strongly connected components (SCCs). OBFR-MP is a divide-and-conquer algorithm like FB [40] (see Section 4.4), but processes the graph in layers. We compared the performance of the OBFR-MP implemented in HipG against a highly-optimized C/MPI version of this program used for performance evaluation in [18] and kindly provided to us by the authors. The HipG version was implemented to resemble the C/MPI version: the data structures used and messages sent are the same. This is because here we are not interested in the speedup of the decomposition algorithm, which may vary depending on the input [18]; rather, we want to see the difference in performance between an optimized C/MPI version and HipG version of the same application.

The experiments were performed on the DAS-3 [50] cluster, which has less memory than DAS-4, but allows for a richer performance analysis. DAS-3 consists of 74 dual dual-core 2.4 GHz AMD Opterons with 4 GB of memory per compute node. The compute nodes are interconnected with 10G-Myrinet and 1G-Ethernet. We compare HipG against (i) ‘MX’, which denotes a low-latency virtually-unbeatable MPI implementation from Myrinet tied to the network interface; (ii) ‘OM’, which denotes OpenMPI, a newer socket-based implementation of MPI, running over Myrinet; and (iii) ‘P4’, the standard implementation of MPI, over Ethernet.

The experiments use 4–64 compute nodes, with one worker per node. We tested OBFR-MP on synthetic graphs called $L_mL_mT_n$, which are in essence trees of height $n$ of SCCs, such that each SCC is a lattice $(m + 1) \times (m + 1)$.
Distributed processing of large graphs

1) An \( L_mL_mT_n \) graph has thus \( (2^{n+1} - 1) \) SCCs, each of size \((m + 1)^2\). The performance of the OBFR-MP algorithm inherently depends on the SCC-structure of the input graph, which is clearly visible in the MX, OM and P4 columns of Table 4.2. We used three graphs: one with a small number of large SCCs, \( L_{487}L_{487}T_5 \); one with a large number of small SCCs, \( L_{10}L_{10}T_{16} \); and one that balances the number of SCCs and their size, \( L_{60}L_{60}T_{11} \). Each graph contains a little over \( 15 \cdot 10^6 \) nodes and \( 45 \cdot 10^6 \) edges.

The C/MPI application running over MX is the fastest, as it has the smallest software stack. HipG performs, on average, 1.8 times slower than MX, but the most fair opponents for HipG are OM and P4, which have a similar (deeper) socket-based software stack. Table 4.2 is summarized in Figure 4.9, where execution times are scaled against MX and P4. On average, HipG is 2.0 times faster than OM on Myrinet, and 2.5 times faster on Ethernet. Most importantly, the speedup or slowdown of HipG follows the speedup or slowdown of the C/MPI application run over MX, which suggests that the overhead of HipG will not explode for larger problem sizes.

Memory utilization

In graph algorithms, more important than speedup is memory efficiency (see further discussion). In a HipG worker, memory is divided between the graph, the communication buffers and the memory allocated explicitly by the user. On a 64-bit machine, a graph node uses 80 bytes in Visitor and on average 1 KB in BFS, including the edges and all overhead. Table 4.1 presents the maximum heap size used per-worker. It remains almost constant for Visitor and BFS, which is expected, as the graph size is doubled when we double the number of workers. BFS uses in general more memory than Visitor, because it stores a queue of nodes (see Figure 4.3).

In SpinJadi, graph nodes are larger: each contains a byte array that represents a state. The size of this array depends on the input program: for the two tested protocols it is about 30 bytes; for TCRsig29, it is about 90 bytes. This difference is because the biological model contains a large number of small concurrent components—29 (originally 40) protein species—while the
BEEM models contain 5–6 large components (the processes). In fact, this is a typical difference between models designed by humans, and models evolved by nature.

**Metrics for distributed graph algorithms**
The results in this section do not aim to prove that we obtained the most efficient implementations of the **Visitor**, BFS, MAP or OBFR-MP algorithms. When processing large-scale graphs, the speedup less important than being able to store the graph in memory and process it in acceptable time. We note that the sizes of graphs tested with HipG are of the order of the largest existing real-life graphs mentioned in Section 4.1. We aimed to show that large-scale graphs *can* be handled by HipG and satisfactory performance can be obtained with little coding effort, even for complex on-the-fly or hierarchical graph algorithms.

### 4.8 Summary and conclusions

In this chapter we propose (1) HipG, a model and a distributed framework that allows users to code, with little effort, parallel graph algorithms; and (2) SpinJadi, a distributed enumerative on-the-fly LTL model checker implemented using HipG. The key idea in our graph framework is to expose each graph node as an object with customizable data and sequential methods, and to provide a unified interface to executing methods on local and non-local graph nodes. Fine-grained recursive computations implemented this way can be controlled using synchronizers. HipG parallelizes such an application automatically. Using HipG we obtained elegant and short implementations of several published graph algorithms, good memory utilization and performance, as well as out-of-the-box portability.

An important feature of HipG is that it allows algorithms to execute on graphs generated on-the-fly; to create SpinJadi, we combined this feature with two model checking algorithms: state space generation and accepting cycle detection [13]. We tested the model checker on large models from the BEEM repository [15], and on biological examples from the GInSim repository [17] that we translated into Promela.

Of the verification methods presented in this dissertation, enumerative model checking is the most widely applicable and can handle the most general properties. Its main limitation is that it requires vast amounts of memory to store state spaces, which we alleviate by splitting the state space between memories of multiple computers. The major drawback of this approach—is the need to parallelize verification algorithms, including any state space reduction methods. In the future, we would like to apply HipG to parallelize more graph algorithms, in particular in the context of distributed model checking, for example supporting modeling languages beyond Promela, and implementing reduction techniques such as Partial Order Reduction [51]. We would also like to improve speedup by using better graph partitioning methods, such as one proposed in [5]. It would also be interesting to use external-memory [29] or semi-external memory (flash drives) [30, 52] to store portions of a graph during computation. HipG
currently does not address fault-tolerance, which could be implemented by freezing the computation and checkpointing, or with a distributed snapshot algorithm, for example the one by Lai-Yang [45].

4.9 Bibliography


[4] J. Alpert and N. Hajaj. We knew the web was big..., 2008. URL: http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html.


4.9. Bibliography


[37] Resource description framework. URL: http://www.w3.org/RDF.


4.9. Bibliography


Distributed search for terminal strongly connected components

5.1 Introduction

In Chapter 4, we verified biological systems using model checking, which we posed as a graph algorithm. Generally, any graph algorithm could be applied to state spaces to understand the behavior of the system at hand. One such custom verification algorithm in the context of biology is decomposition of a state space into strongly connected components (SCCs) [1–3]. In a directed graph, an SCC [4] is a maximal subgraph that contains a path from any vertex to any other vertex. Of particular importance are terminal SCCs (TSCCs), i.e. SCCs from which no other component can be reached; in other words, a TSCC is a leaf in a graph in which each SCC is contracted into a single vertex. An SCC in a state space of a biological model, e.g. a cell, can represent an irreversible state of the system; a TSCC corresponds to a steady state, or a state of terminal differentiation (when the cell stops specializing), also called an ‘attractor’ [1, 2]. Besides biology, TSCCs are useful in many other domains (see Section 5.2).

In this chapter, we address the open challenge of efficiently finding TSCCs in large graphs. In a sequential setting, TSCCs are best found with an efficient SCC-decomposition algorithm based on depth-first search (DFS), such as the one by Tarjan [5]. Several parallel SCC-decomposition algorithms have been proposed [6–8], based on the computation of reachable vertices from a given vertex, which—unlike DFS [9]—is an operation that can be parallelized efficiently. Another inspiring algorithm, CH, introduced by Orzan [10], finds all SCCs by coloring vertices and removing SCCs of vertices with specific properties (see Section 5.2 and Section 5.7 for more details). In all cases, if only TSCCs are needed, the parallel SCC-decomposition algorithms do unnecessary work.

In this chapter we present TSCCdc, a novel parallel algorithm specifically designed to find TSCCs. The key is that the search for TSCCs is expressed
Distributed search for terminal strongly connected components

... recursively, and thus can be solved with a divide-and-conquer graph algorithm. An input graph is split using reachability computations into subgraphs which cannot be ‘crossed’ by SCCs. By construction (see Section 5.5) one of the subgraphs can be discarded; one is an SCC, which is checked for being terminal; and two subgraph are searched recursively in parallel.

Even on unfavorable graphs, we found TSCCdc on average more than twice faster than OBFR-MP [7], a state-of-the-art SCC-decomposition algorithm, and a factor of ten faster than the CH algorithm adapted to only find TSCCs (see Section 5.7). We also studied real-life examples: graphs representing a biological system of human blood cells [3]. In one case our algorithm obtained a 200-fold speedup, and in two cases it was the only one able to solve the problem at all.

All algorithms in this chapter, including the competing ones, were implemented using HipG, our distributed framework for distributed graph algorithms, described in Chapter 4. Thanks to HipG, we obtain code that is elegant and easy to modify, as well as efficient.

Additionally, we observe that applying TSCCdc to a transposed graph finds all leading SCCs, i.e. SCCs which cannot be reached from any other SCCs.

The remainder of this chapter is organized as follows. In Section 5.2 we discuss the related work. Preliminary definitions are briefly introduced in Section 5.3. Section 5.4 explains the relation between reachable sets and SCCs, and provides a correctness proof for the TSCCdc algorithm explained in Section 5.5. Section 5.6 describes the implementation of our algorithm, which is evaluated in Section 5.7. We conclude with Section 5.8.

**Brief description of TSCCdc**

The full description of the algorithm can be found in Section 5.5. Applied to a graph with vertices \(V\), TSCCdc first computes a set \(F\) of vertices reachable in \(V\) from a random pivot in \(V\). Next, it computes a set \(B\) of vertices backward reachable in \(V\) from \(F\). Finally, TSCCdc computes \(C\), the SCC of the pivot, using a backward reachability search within \(F\). We observe that this way the set \(V\) was divided into four nested graphs: \(C \subseteq F \subseteq B \subseteq V\). By construction, \(B \setminus F\) cannot contain TSCCs (see Section 5.4) and is discarded; the SCC \(C\) is terminal if and only if \(F = C\); the two remaining subgraphs, \(V \setminus F\) and \(F \setminus C\), if non-empty, are solved with TSCCdc recursively in parallel.

While the idea of recursive graph computation is identical to that proposed by Fleischer et al in [6], the algorithm in [6] differs from TSCCdc in how the graphs are divided into subgraphs, how the subgraphs are handled. Most notably, \(B\) and \(C\) are computed differently: we compute \(B\) as a backward reachability from the result of another reachability search, rather than from the pivot—this way we discard a larger \(B \setminus F\) set. Another difference with [6] is that we discard the result of the backward reachability search instead of solving it in parallel.

**5.2 Related work**

Besides biology, a major area where TSCCs are useful is model checking [11], for example checking safety properties [11, Chapter 3], efficient state space
5.3. Preliminaries

In this chapter we consider directed graphs (the notion of strongly connected components does not make sense for undirected graphs). A transpose of a directed graph is the same graph with the edges reversed. A directed graph is strongly connected if there exists a path between any pair of its vertices.
A maximal strongly connected subgraph of a graph \( G \) is called a **strongly connected component** (SCC) of \( G \). Each vertex belongs to exactly one SCC, so the decomposition of a graph into SCCs is well-defined and unique.

A *quotient graph* is a graph in which each SCC is contracted into a single vertex. There is an edge \( C_1 \rightarrow C_2 \) between vertices in the quotient graph if there exists an edge from a vertex in \( C_1 \) to a vertex in \( C_2 \). The quotient graph is directed and acyclic. Figure 5.1 shows an example quotient graph. In the quotient graph, if a component has no incoming edges, it is *leading*. If it has no outgoing edges, it is called a *terminal* SCC (TSCC). The example graph in Figure 5.1 has six SCCs: three are trivial (a single vertex), two are leading and two are terminal.

If there is a directed path from vertex \( v \) to vertex \( w \) (\( v \leadsto w \)), we say that \( w \) is *reachable* (or forward-reachable) from \( v \), and that \( v \) is *backward-reachable* from \( w \). In particular, any vertex is both reachable and backward-reachable from any other vertex within an SCC. By performing a forward (backward) reachability search from a given pivot vertex \( v \), we mean computing the set of all vertices forward (backward) reachable from \( v \). The vertex from which a reachability search is initiated is commonly referred to as a pivot vertex.

### 5.4 Reachability versus strongly connected components

In this chapter we use two procedures, \( \text{Fwd} \) and \( \text{Bwd} \), that compute sets of reachable vertices in a directed graph \( G = (V_G, E) \). Let \( V \) be a set of vertices in \( V_G \). Then \( \text{Fwd}(P, V) \) computes all vertices in \( V \) reachable from some vertex in \( P \subseteq V \). Similarly, \( \text{Bwd}(P, V) \) computes all vertices in \( V \) backward-reachable from \( P \). Typically, \( P \) contains a single pivot vertex.

Computing reachable vertices carries important implications about the SCCs. First, no SCC can 'cross' a border of a set computed with \( \text{Fwd} \) or \( \text{Bwd} \) [6]:

**Lemma 5.1.** Let \( P \subseteq V \) and \( F = \text{Fwd}(P, V) \). Any SCC of \( V \) must lie either completely within \( F \) or completely within \( V \setminus F \). This is also true if \( F = \text{Bwd}(P, V) \).

**Proof.** Consider a set of vertices lying partly within \( F \) and partly outside \( F \). For \( F = \text{Fwd}(P, V) \), the outside part is not reachable from the inside part, so this
is not an SCC. Likewise, for $F = \text{Bwd}(P, V)$, the inside part is not reachable from the outside part.

As a consequence, given a set returned by $\text{Fwd}$ or $\text{Bwd}$, we can search for SCCs independently within and outside of this set.

We can also use reachability searches to demarcate SCCs that contain a given vertex $v$ as an intersection $\text{Fwd}(v, V) \cap \text{Bwd}(v, V)$. This fact is the cornerstone of the SCC-decomposition algorithm proposed in [6]. Equivalently, $V$ can be changed to $F$ in $\text{Bwd}$ in this formulation:

**Lemma 5.2.** Let vertex $v \in V$ and $F = \text{Fwd}(v, V)$. The SCC $C$ that contains $v$ is $\text{Bwd}(v, F)$.

**Proof.** Consider a vertex $w$ in the SCC that contains $v$, i.e. there exist paths $v \leadsto w$ and $w \leadsto v$. Clearly, all vertices on both paths belong to $F$. Therefore, $\text{Bwd}(v, F)$ will discover them.

The $\text{Fwd}$ and $\text{Bwd}$ operations are used in this chapter to find TSCCs. First, note that an SCC is terminal if it is equal to its forward-reachable set. Second, combining the $\text{Fwd}$ and $\text{Bwd}$ computations we can find regions with and without TSCCs. Namely, given any SCC, the components ‘upstream’ of it in the quotient graph cannot be terminal. Even stronger, the same is true if we consider SCCs ‘upstream’ of a set of SCCs, namely:

**Lemma 5.3.** Let $v \in V$, $F = \text{Fwd}(v, V)$ and $B = \text{Bwd}(F, V)$. No SCCs in $B \setminus F$ can be terminal.

**Proof.** This lemma is illustrated in Figure 5.2. Note that $F \subseteq B$. Applying Lemma 5.1 twice, any SCC must lie entirely within one of the three sets: $V \setminus B$, $B \setminus F$ or $F$. Since $B \setminus F$ is a result of a backward reachability search, from any SCC in $B \setminus F$, an SCC within $F$ can be reached.

This lemma is the foundation of our TSCC-search: our algorithm repeatedly generates such $F$ and $B$ sets and discard the $B \setminus F$ set.

![Figure 5.2: Illustration of Lemma 5.3: $F = \text{Fwd}(v, V)$ and $B = \text{Bwd}(F, V)$; there are no TSCCs in $B \setminus F$.](image-url)
5.5 The TSCCdc algorithm

In this section we present TSCCdc, a parallel algorithm to find all terminal SCCs in a directed graph. TSCCdc is a divide-and-conquer algorithm: using reachability searches we partition a graph into several regions and handle each region independently in parallel. This idea is inspired by the SCC-decomposition algorithm [6].

The pseudocode of TSCCdc is displayed in Figure 5.3, along with an illustration. The input is a directed graph $G = (V, E)$. If it is empty, it contains no SCCs. Otherwise, we pick a pivot $v$ at random from $V$. First, we construct the set $F = Fwd(v, V)$ of vertices in $V$ reachable from $v$. Next, we compute the set $B = Bwd(F, V)$ of vertices from which there exists a path to some vertex in $F$. The sets $F \subseteq B$ divide $V$ into three subsets, $V \setminus B$, $B \setminus F$, and $F$, and any SCC in $V$ is entirely contained in one of them (Lemma 5.1). Most importantly, there can be no TSCCs in $B \setminus F$ (Lemma 5.3), so we no longer take it into account. Note that, unlike [6], we compute $B$ as a backward reachability search of an entire set of pivots $F$, rather than only $v$; this way we increase the size of $B \setminus F$, which can be discarded.

To find TSCCs in $F$, we need the SCC $C$ that contains $v$, which we compute as the intersection of forward and backward reachability searches from $v$ (Lemma 5.2). If $C$ equals $F$, we report it as a new TSCC. Next, we search for TSCCs within $V \setminus B$ and within $F \setminus C$; the computations in $V \setminus B$ and $F \setminus C$ are independent, so they can be performed in parallel. The efficiency of our procedure relies on the fact that reachability can be efficiently parallelized.

TSCCdc terminates because the graph is fixed and the set of vertices removed in each iteration is non-empty, i.e. $v \in C, F, B$. The worst-case time complexity of TSCCdc is $O((|V| + |E|)^2)$, for example for a ‘line’ graph with, where we remove 1-element SCC at a time using three reachability computations of cost $O(|V| + |E|)$.

5.6 Implementation

We implemented TSCCdc on a distributed-memory machine using HipG, our framework for writing distributed graph algorithms (Chapter 4). The algorithm in Figure 5.3 does not preclude implementation on a shared-memory machine, but our intention is to handle large graphs, for which we need more memory; therefore, besides being task-parallel, the algorithm that we report on in this chapter is also data-parallel. HipG was described in detail in Chapter 4; in the current section, before we discuss TSCCdc itself, we briefly reiterate some of the details of the framework relevant to the implementation of TSCCdc.

**HipG**

HipG is written in Java, and so is the code in this section. The framework allows expressing graph algorithms using high-level concepts of vertices and edges, which results in easily readable and maintainable code. Each worker is assigned a portion of the vertices of the input graph and executes methods on vertices that it owns. The basis of the TSCCdc algorithm is computation of
5.6. Implementation

```plaintext
TSCCdc (V) {
  if (V ≠ ∅) {
    v = pick a random pivot in V
    F = Fwd (v, V)
    parallel {
      B = Bwd (F, V)
      C = Bwd (v, F)
    }
    if (C == F) report C to be a TSCC
    parallel {
      TSCCdc (F \ C)
      TSCCdc (V \ B)
    }
  }
}
```

Figure 5.3: TSCCdc: a divide-and-conquer graph algorithm to search for TSCCs. With the reachability searches in Lemma 5.3 we decompose the graph \( G = (V, E) \) into \( V \setminus B, B \setminus F, \) and \( F, \) from which we further tease out \( C, \) the SCC that contains pivot \( v. \)

Distributed reachability. HPG allows to execute methods on any vertex, even non-local, in which case HPG translates the calls into asynchronous messages. Using this feature we implemented the reachability searches that compute \( F, B \) and \( C. \) For example, Figure 5.4 shows a snippet of the forward reachability search that computes \( F. \) Such a search is initiated from \( v \) with a single call to \( v.fwd(). \) A call to \( fwd() \) may generate calls to \( fwd() \) on other vertices on other workers, which may initiate more \( fwd() \) calls, and so on. To detect that all \( fwd() \) calls have been completed and none are in transit, we use \( barrier() \), which is in HPG based on the distributed termination detection algorithm by Safra [28]. Note that the code in Figure 5.4 is only a snippet: in the remainder we assign the vertex to \( F \) and flag it as visited (see the second part of this section for more algorithm-specific implementation details).

TSCCdc consists of a tree of tasks (synchronizers) running independently in parallel, of which each executes reachability searches and barriers (described above). Besides computing on vertices, tasks spawn new tasks to solve sub-problems. The runtime system provides exclusive execution of the synchronizers, so that no locks or thread synchronization is necessary in the user code, even when accessing the synchronizer’s data. A snippet of the TSCCdc synchronizer is displayed in Figure 5.5. Unlike the code in Figure 5.4, which is executed by the vertex owner, the code in Figure 5.5 is executed by all workers. Together they spawn a new task: logically, a spawn spawns a single new task, which is represented at each worker. The runtime system takes care of starting, executing and terminating the synchronizers.
Distributed search for terminal strongly connected components

```java
class TscLocNode extends LocalNode {
    public final void fwd(TSCCdc) {
        if (inV && !visited)
        ...
        for (int j = 0; hasNeighbor(j); ++j)
            neighbor(j).fwd(tscc);
    }
}
```

Figure 5.4: TSCCdc: computing F.

```java
class TSCCdc extends Synchronizer {
    public void run() {
        ...
        if (globalFsize == globalCsize)
            TSCCs.addComponent(...);
        else spawn(new TSCCdc(g, F));
    }
}
```

Figure 5.5: Spawning a new task.

**TSCCdc**

TSCCdc generates many sets of vertices: each set is assigned a unique identifier, and a vertex belongs to the set if its field id (not shown) equals that of the set. As a consequence, each vertex belongs to exactly one set at a time. In addition, to enable iteration over elements of a set, sets are stored explicitly as partitioned lists (an example of this was given in Figure 4.3).

The input of a TSCCdc synchronizer is a set of vertices V. Then it proceeds as follows (see Figure 5.3). First, a random pivot v is selected from V. This is realized with a global reduce operation, which HipG supports [29]. Next, the F reachability search is performed. Within fwd(), a vertex adds itself to the new set F. After F has been built completely, two backward reachability searches are performed: computation of C starting at v and computation of B starting at all elements of F. When all reachability searches have terminated, a TSCC is checked and a new task started as in Figure 5.5, with B \ F and the remainder of V as inputs.

### 5.7 Evaluation

We evaluated our algorithm on the DAS-4/VU cluster [30] on up to 64 compute nodes. Each compute node runs Linux (CentOS, version 6), has 2 × 4 Intel cores of speed 2.4 GHz, and is equipped with 24 GB of memory. The cluster interconnect is 4xQDR InfiniBand. As a Java Virtual Machine we used Oracle’s HotSpot version 1.6 [31]. We used three kinds of graphs:

- **ToL (n, m)**, a ‘tree of lattices’ illustrated in Figure 5.6(a), a binary tree of height n with edges directed toward leaves, such that each tree vertex represents an SCC. Each SCC is a lattice $m \times m$ with columns and rows looped. This graph has a total of $(2^{n+1} - 1)m^2$ vertices. The tree is the quotient graph, of which the $2^n$ leaves are TSCCs.

- **LoL (n, m)**, a ‘lattice of lattices’ illustrated in Figure 5.6(b), an $n \times n$ mesh of SCCs with edges directed toward the south-east corner. Like in ToL, each SCC is an $m \times m$ lattice with rows and columns looped. A LoL (n, m) graph has $n^2m^2$ vertices. The $n$-lattice is the quotient graph, with the south-east corner as the only TSCC.

- **BioPNSS (n)**, a state space of a Petri net [18] representing a biological system of differentiation of human blood cells [3]. The number n is
5.7. Evaluation

Figure 5.6: Synthetic graphs: a tree (a) and a lattice (b) of looped lattices.

the network ‘granularity’, i.e. the number of values that any variable can take—for example $n = 2$ would mean a Boolean network. The size of the graph thus grows exponentially with $n$.

We compare TSCCdc against two algorithms: OBFR-MP and TSCCCh. OBFR-MP is a state-of-the-art SCC-decomposition algorithm [7]. It splits a rooted graph into OBF-layers. If a graph is not rooted it first decomposes it into rooted chunks. The O-layer consists of vertices without predecessors, the B-layer is the backward-reachable set of the O-layer, and the F-layer contains successors of the B-layer. The B-layers are handled recursively by the same algorithm.

While the ideas behind OBFR-MP are fairly similar to those in this chapter, the concept of CH [10] is diametrically different from all reachability-based

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>SCCs</th>
<th>SCC size</th>
<th>TSCCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToL(0, 20000)</td>
<td>400 · 10^6</td>
<td>800 · 10^6</td>
<td>1</td>
<td>400 · 10^6</td>
<td>1</td>
</tr>
<tr>
<td>ToL(5, 2500)</td>
<td>394 · 10^6</td>
<td>788 · 10^6</td>
<td>63</td>
<td>6 · 10^6</td>
<td>32</td>
</tr>
<tr>
<td>ToL(10, 500)</td>
<td>512 · 10^6</td>
<td>1024 · 10^6</td>
<td>2047</td>
<td>250000</td>
<td>1024</td>
</tr>
<tr>
<td>ToL(14, 120)</td>
<td>472 · 10^6</td>
<td>944 · 10^6</td>
<td>32767</td>
<td>14400</td>
<td>16384</td>
</tr>
<tr>
<td>LoL(8, 2500)</td>
<td>400 · 10^6</td>
<td>800 · 10^6</td>
<td>64</td>
<td>6 · 10^6</td>
<td>1</td>
</tr>
<tr>
<td>LoL(45, 500)</td>
<td>506 · 10^6</td>
<td>1013 · 10^6</td>
<td>2025</td>
<td>250000</td>
<td>1</td>
</tr>
<tr>
<td>LoL(180, 140)</td>
<td>635 · 10^6</td>
<td>1270 · 10^6</td>
<td>32400</td>
<td>19600</td>
<td>1</td>
</tr>
<tr>
<td>BioPNSS(5)</td>
<td>49 · 10^6</td>
<td>537 · 10^6</td>
<td>11 · 10^6</td>
<td>≤ 960</td>
<td>2</td>
</tr>
<tr>
<td>BioPNSS(6)</td>
<td>362 · 10^6</td>
<td>3990 · 10^6</td>
<td>≥ 2</td>
<td>≥ 1</td>
<td>2</td>
</tr>
<tr>
<td>BioPNSS(7)</td>
<td>1977 · 10^6</td>
<td>21750 · 10^6</td>
<td>≥ 2</td>
<td>≥ 1</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 5.2: Run times for ToL graphs (seconds). 'M' means out of memory.

<table>
<thead>
<tr>
<th>p</th>
<th>ToL(0, 20000)</th>
<th>ToL(5, 2500)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TSCC dc</td>
<td>OBFR -MP</td>
</tr>
<tr>
<td>8</td>
<td>128.1</td>
<td>146.0</td>
</tr>
<tr>
<td>16</td>
<td>80.5</td>
<td>86.4</td>
</tr>
<tr>
<td>32</td>
<td>47.6</td>
<td>53.0</td>
</tr>
<tr>
<td>64</td>
<td>29.2</td>
<td>33.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>ToL(10, 500)</th>
<th>ToL(14, 120)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TSCC dc</td>
<td>OBFR -MP</td>
</tr>
<tr>
<td>8</td>
<td>255.9</td>
<td>369.5</td>
</tr>
<tr>
<td>16</td>
<td>186.0</td>
<td>241.4</td>
</tr>
<tr>
<td>32</td>
<td>155.9</td>
<td>189.5</td>
</tr>
<tr>
<td>64</td>
<td>120.2</td>
<td>128.4</td>
</tr>
</tbody>
</table>

Table 5.3: Run times for LoL graphs (seconds). 'M' means out of memory.

<table>
<thead>
<tr>
<th>p</th>
<th>LoL (8, 2500)</th>
<th>LoL (45, 500)</th>
<th>LoL (180, 140)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TSCC dc</td>
<td>OBFR -MP</td>
<td>TSCC ch</td>
</tr>
<tr>
<td>8</td>
<td>87.6</td>
<td>261.0</td>
<td>M</td>
</tr>
<tr>
<td>16</td>
<td>58.5</td>
<td>152.5</td>
<td>M</td>
</tr>
<tr>
<td>32</td>
<td>37.0</td>
<td>114.2</td>
<td>516.8</td>
</tr>
<tr>
<td>64</td>
<td>12.9</td>
<td>69.2</td>
<td>253.6</td>
</tr>
</tbody>
</table>

Table 5.4: Run times for BioPNSS graphs (seconds). 'M' means out of memory; 'T' is timeout (2 hours).

<table>
<thead>
<tr>
<th>p</th>
<th>BioPNSS(5)</th>
<th>BioPNSS(6)</th>
<th>BioPNSS(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TSCC dc</td>
<td>OBFR -MP</td>
<td>TSCC ch</td>
</tr>
<tr>
<td>8</td>
<td>29.6</td>
<td>1013.1</td>
<td>438.1</td>
</tr>
<tr>
<td>16</td>
<td>13.7</td>
<td>1162.2</td>
<td>183.6</td>
</tr>
<tr>
<td>32</td>
<td>8.4</td>
<td>1475.0</td>
<td>72.5</td>
</tr>
<tr>
<td>64</td>
<td>5.0</td>
<td>2505.2</td>
<td>33.4</td>
</tr>
</tbody>
</table>
algorithms: the key idea is that the vertices are colored with colors that increase along edges (see Section 5.2). For the purpose of evaluation, we adapted CH to TSCCch, which searches only for TSCCs. Let us consider the SCCs found by a single iteration of CH: observe that in particular all leading SCCs were found. It can be determined whether the component is leading or not during the reachability phase; namely, if a component has an incoming edge from a vertex of another color, that component is not leading. This algorithm finds all leading components, so, applied to the transposed graph, finds all TSCCs. The TSCCch algorithm accounts for all these observations: it is a single iteration of CH, applied to the transposed graph, and with the reachability phase augmented with checking for edges incoming from vertices with a different color.

Table 5.2 and Table 5.4 show run times of the TSCCdc, OBFR-MP and TSCCch algorithms applied to the graphs listed in Table 5.1. The time spent reading the graph is not included in the measurements. In all runs, vertices were randomly assigned to processors. All algorithms pick pivots truly at random, as compared to taking the first one found, which may bias run times. For the same reason, TSCCch randomizes initial colors of vertices.

Figure 5.7 displays the run times from Table 5.2 and Table 5.4, averaged per graph and normalized to TSCCdc. We show only the smallest BioPNSS graph, because the competing algorithms fail on the large graphs.
taking the BioPNSS graphs into account, TSCCdc is on average more than twice faster than OBFR-MP, and faster than TSCCch by a factor close to 10. On the BioPNSS (5) graph, TSCCdc is almost 200 faster than OBFR-MP, and 7 times faster than TSCCch. We also observe that TSCCdc performs better—in comparison to OBFR-MP—on the LoL graphs than on the ToL graphs, because of larger backward-reachable sets in the LoL graphs (sub-lattices of $n \times n$ lattices compared to path too root in a tree of height $n$). TSCCdc is slower in one case only; namely, for ToL (14, 120), which is the graph with a structure most unfavorable to TSCCdc: large $F$ sets and small $B \setminus F$ sets. Most notably, for the BioPNSS (6) and BioPNSS (7) examples (Table 5.4), TSCCdc was the only algorithm to solve the given problems at all, as it was able to quickly discard large $B \setminus F$ sets.

Note that TSCCch often runs out of memory—this is caused by the buffering space needed to handle an enormous volume of communication that results from recoloring vertices (a single recolored vertex may cause recoloring of the entire graph). Indeed, averaging all completed runs of TSCCch, a vertex was recolored 38 times during the execution of the algorithm. This problem is exacerbated in graphs with large SCCs like ToL (0, 20000), as all vertices in an SCC must end up with the same color. Contrastingly, graphs with a large number of small SCCs, such as ToL (14, 120), are likely to have many ‘heads’, which means less recoloring. In such a case TSCCch is a clear winner, provided it has enough memory—in our experience, the memory needed for buffering must at least be a small multiple of the graph size.

Figure 5.8 shows the speedup of the TSCCdc algorithm per input graph. On graphs with large and medium SCCs, the efficiency of TSCCdc on 64 processors is about 58%; the efficiency on graphs with small SCCs, i.e. ToL (14, 120) and LoL (180, 140) is about 38%. Poorer scalability on ToL graphs results from the fact that the backward-reachable sets are small in such a tree, and many small forward-reachability searches must be performed in branches.

We observed that scalability of the algorithms considered here greatly depends on the average length of messages sent. The communication pattern in these algorithm is as follows. Each processor communicates with all other
5.8 Summary and conclusions

processors, and the traffic consists almost entirely of small pieces of work sent to remote vertices. Each piece of work is typically of size 16–20 bytes, so each processor aggregates these small messages per destination. Aggregation (also called ‘message combining’) must be balanced for good performance: while too aggressive aggregation may delay progress of the algorithm, too weak aggregation leads to short messages and message start-up time dominates the run time. In our experience good performance was obtained when message length was at least 0.5–1 KB. In case of graphs with many small SCCs, such as ToL (14, 120), the average message length drops to 0.05–0.2 KB for TSCCdc, but for the same graph it is an acceptable 0.5–1 KB in OBFR-MP, which performs larger reachability searches in parallel.

We point out that, when processing large graphs, the first objective is to ensure the graphs can be processed at all in an acceptable time. Good speedup is a secondary objective. In case of our algorithm, not only was TSCCdc the only one able to solve the BioPNSS (6) and BioPNSS (7) examples, but also it sustains efficiency of 40 – 60% on 64 compute nodes. This means that both objectives were met.

5.8 Summary and conclusions

We have presented a new parallel algorithm, TSCCdc, for finding TSCCs in large graphs. It is a divide-and-conquer algorithm, which, at each recursion level, computes a backward-reachability search of a set obtained by a forward-reachability search from a random vertex. The algorithm is memory-bound, so our implementation targets a distributed-memory architecture. The algorithm and its competitors in this chapter were implemented using HipG, our high-level framework for implementing distributed graph algorithms.

We conclude that a user wanting to find TSCCs in a large graph should use TSCCCCH, a version of the CH [10] algorithm adapted to only find TSCCs (see Section 5.7)—if the graph contains many small SCCs and the size of the memory is at least several times larger than the graph. In all other cases, our TSCCdc algorithm offers a performance improvement of up to several orders of magnitude over a full SCC-decomposition. In the real-life biological model of blood cells formation [3], it was the only algorithm that was able to compute TSCCs.

Barnat et al [7] reuse a model checking technique called OWCTY, for fast removal of trivial SCCs; an idea for future work is to add this technique to TSCCdc. It would also be interesting to apply Schudy’s optimization [27] (see Section 5.2) to our algorithm.

5.9 Bibliography


5.9. Bibliography


Summary and conclusions

Executable models are the most important tool of systems biology: they are used to unambiguously encode our understanding of complex biological processes, and they allow to conduct experiments otherwise technically impossible or unethical. Examples of executable models include differential equations, Petri nets, Boolean and qualitative networks, stochastic systems, and many others. In order to make model-based predictions about biology, the models should faithfully represent nature; to this end, the models must be verified: checked against the known biological evidence.

In this dissertation we investigated how to verify large discrete models of biological systems. Such systems typically consist of a large number of concurrently-executing small components. The main challenge when verifying them is state explosion: exponential growth of the state space with the number of concurrent components in the system. Therefore, the main focus of this work is scalability: ability to handle systems with very large state spaces. In order to achieve scalability, we use existing techniques and propose new ones thus making contributions to the fields of high-performance computing and model checking.

6.1 Summary of the thesis

In Chapter 2 we studied Monte Carlo simulations. In this method, a system is analyzed by performing and examining a large number of simulations; intuitively, a population of animals is emulated undergoing the studied biological process. We applied this approach to our model of cell fate determination during formation of a vulva (an egg-laying organ) in the C. elegans worm; the size of the state space of this model is in the order of $2^{713}$. For each of the 64 genetic perturbations of our model, we performed 5000 simulations. Besides aggressive optimization of a single simulation, we parallelized the Monte Carlo experiments for a cluster of computers, which resulted, on a distributed machine with 256 cores, in reducing the time needed to run the entire suite of verification experiments to less than an hour.
While simulations may not reach all corners of a state space, formal methods, i.e. analyzing a model as a computer program, are able to check all states or all paths of a system. One such method is abstract interpretation, which allows to prove a property about a system by interpreting only parts of it relevant to the property under consideration. In Chapter 3, we proposed BioCHECK, an efficient procedure to prove stabilization (reaching a unique fixpoint) of systems. The tool proves stabilization by building the global liveness property from a chain of small liveness properties, which are fast to prove. BioCHECK achieves scalability by applying state space exploration only locally to small pieces of the system rather than the entire system as a whole. We used it to prove stabilization of a 3-D mesh of 200x500x5 mammalian skin cells; the state space of this model contains \(2^{6\text{mln}}\) reachable states.

In Chapters 4 and 5, we treated a state space as a large sparse graph, which has to be split between multiple machines to mitigate the state explosion problem; an important consequence of this approach is that verification algorithms need to be parallelized. Chapter 4 introduced HipG, a high-level framework for writing such distributed graph algorithms. The key idea in HipG is that a user expresses a graph algorithm by defining data stored by a vertex, as well as the vertex' methods. The framework allows to seamlessly execute methods on any graph vertex, local or remote. HipG parallelizes the graph application automatically and handles the details of execution on a distributed machine.

Using HipG we implemented SPINJADI, a distributed enumerative model checker, which explores a state space in an on-the-fly fashion: it starts with an empty graph, explores the system's initial state, its successors, the successors of the successors, and so on, until a bug is found or the state space exhausted. Properties of infinite executions are checked using an on-the-fly cycle detection algorithm by Brim et al. Using SPINJADI, we checked two mutual exclusion protocols, as well as a biological model of T-cell activation during an immune response.

In Chapter 5 we introduced TSCCdc, an efficient distributed algorithm to find TSCCs in large graphs. In biology, TSCCs correspond to states of terminal differentiation (when a cell stops specializing), or to steady states. TSCCdc is a parallel divide-and-conquer graph algorithm: using reachability computations, a graph is split into four independent subgraphs, which cannot be 'crossed' by SCCs, and so can be searched recursively in parallel. We found TSCCdc was the only algorithm able to process our case study: a model of human haematopoietic (blood) cells, of size in the order of \(2^{34}\) of vertices and edges.

### 6.2 Summary of contributions

The core contributions of this dissertation are as follows.

- We introduced a new approach using parallel Petri nets to modeling in biology, which we used to create a model of vulval development in *C. elegans*—this model includes two previously published but not modeled hypotheses about the process.
6.3. Conclusions

- We proposed a novel scalable technique to prove stabilization of concurrent systems, and we were the first to verify a large 3D mesh of mammalian skin cells.

- We designed and implemented a new framework to easily write distributed graph algorithms, and used it to create a distributed on-the-fly enumerative model checker.

- We introduced a novel efficient distributed algorithm to find terminal strongly connected components in a large graph, and thus determine the steady states of a biological system.

6.3 Conclusions

The scalable methods described in this dissertation can help users in the verification of real-life systems, enabling holistic insight into biological processes (see Chapter 1). The state explosion problem makes the task of verification of large parallel systems very challenging. In general, we approached it in two ways. First, we use techniques that are independent of state space enumeration: simulations, and abstract interpretation combined with modular reasoning. Second, we execute verification on a parallel computer, which can fit larger state spaces; in particular, we parallelized model checking, a widely-applicable verification method, as well as a customized biology-inspired method basing on TSCCs. Neither of the presented methods is a clear winner; rather, choosing an appropriate technique depends on the system and the properties targeted.

6.4 Future outlook

We have shown that the methods described in this dissertation can be very useful, but as an 'encyclopedia' of scalable verification methods, this thesis is far from complete. For example, it would be interesting to adapt the described methods to systems with probabilities and systems outside molecular biology. The largest systems in this dissertation were solved using BioCheck; in the future we could adapt BioCheck to solve other liveness properties, for example by conjoining liveness with stabilization. Monte Carlo verification is both scalable and widely-applicable, but it is lacking support to estimate coverage and understand causality. In the field of distributed model checking, we would like to parallelize some of the existing state space reduction techniques, and find new ones targeted specifically at biological systems. Finally, we plan to reuse HipG to implement many other custom graph-based verification algorithms.

This dissertation clearly shows that solving large parallel systems involves interdisciplinary research that encompasses fields such as biology, computing, algorithms, theory. Collaboration between scientists of these different fields is essential for further advancements in systems biology. Looking ahead, we predict that verification methods for larger and larger systems will be needed, in order to, eventually, model, check, and execute an entire organism.
Publications

Parts of this dissertation have been published before. The key publications that this thesis is based upon are:


Naar ‘Big Biology’:
Snelle Verificatie van Grootschalige Simultane Systemen

Executeerbare modellen vormen het belangrijkste instrument van de systeembiologie: ze worden gebruikt voor het eenduidig coderen van ons begrip van complexe biologische processen en ze maken het mogelijk experimenten uit te voeren die anders technisch onmogelijk of onethisch zouden zijn. Om model-gebaseerde voorspellingen te kunnen doen over biologie, moeten modellen een getrouwe weergave zijn van de natuur. Hiertoe moeten modellen gecontroleerd worden, d.w.z., worden vergeleken met het beschikbare biologische bewijsmateriaal.

In dit proefschrift onderzoeken we hoe grote discrete modellen van biologische systemen geverifieerd kunnen worden. Dergelijke systemen bestaan doorgaans uit een groot aantal simultane kleine onderdelen. De belangrijkste uitdaging bij het verifiëren hiervan is de toestandsexplosie: de exponentiële groei van de toestandsruimte bij het vergroten van het aantal simultane componenten in het systeem. Daarom is de belangrijkste focus van dit proefschrift de schaalbaarheid: het vermogen om systemen met een zeer grote toestandsruimte te behandelen. Om schaalbaarheid te bereiken gebruiken we deels bestaande technieken, maar stellen we ook enkele nieuwe voor op het gebied van grootschalig rekenen en model verificatie.

In hoofdstuk 2 bestuderen we Monte Carlo simulaties. In deze methode wordt een systeem geanalyseerd door het uitvoeren en onderzoeken van een groot aantal simulaties; in dit geval wordt een populatie van organismen nagebootst tijdens het ondergaan van het bestudeerde biologische proces. We passen deze benadering toe bij ons model over de bepaling van het celtype tijdens de vorming van een vulva (het orgaan dat eitjes produceert) in de worm C. elegans; de grootte van de toestandsruimte van dit model is in de orde van $2^{715}$. Voor elk van de 64 genetische verstoringen binnen ons model voerden we 5000 simulaties uit. Naast agressieve optimalisatie van individuele simulaties paralleliseerden we de Monte Carlo experimenten op een rekencluster van computers. Op een gedistribueerde machine met 256 processorkernen resulteerde dit in een vermindering van de benodigde tijd voor de volledige verzameling van controle-experimenten tot minder dan een uur.
Waar simulaties mogelijk niet alle uithoeken van een toestandsruimte kunnen bereiken, zijn formele methoden, d.w.z. het analyseren van een model in de vorm van een computerprogramma, wel in staat alle toestanden of trajecten in een systeem te controleren. Een dergelijke methode is abstracte interpretatie, die het mogelijk maakt om een eigenschap van een systeem te bewijzen door het interpreteren van slechts de relevante delen hiervan voor de betreffende eigenschap. In hoofdstuk 3 introduceren we BioCheck, een efficiënte procedure voor het bewijzen van stabilisatie (het bereiken van een unieke vast punt) van systemen. Deze applicatie bewijst stabilisatie door de constructie van de globale liveness eigenschap uit een keten van kleinere liveness eigenschappen, die snel zijn aan te tonen. BioCheck bereikt schaalbaarheid door het alleen lokaal doorzoeken van de toestandsruimte op kleine delen van het systeem in plaats van het systeem als geheel. We gebruikten dit om stabilisatie van een 3-D topologie van $200 \times 500 \times 5$ zoogdier-huidcellen te bewijzen; de toestandsruimte van dit model bevat $2^{26} \text{mln}$ bereikbare toestanden.

In hoofdstukken 4 en 5 behandelen we een toestandsruimte als een grote ijle graaf, die moet worden verdeeld over meerdere machines om de toestandsexplosie te beperken; een belangrijk gevolg van deze aanpak is dat de verificatie-algoritmen moeten worden geparalleliseerd. Hoofdstuk 4 introduceert HiPG, een hoog niveau raamwerk voor het schrijven van deze gedeelde graafalgoritmen. De kerngedachte in HiPG is dat een gebruiker een graafalgoritme uitdrukt door het definiëren van de data opgeslagen door een knoop en de methoden die geëxecuteerd kunnen worden op een knoop. Het raamwerk maakt het mogelijk om naadloos methoden op een knoop van de graaf uit te voeren, zowel lokaal als op een andere machine. HiPG paralleliseert de graafapplicatie automatisch en zorgt voor de details van de uitvoering op een gedeelde machine.

Met behulp van HiPG implementeerden we SPINJADI, een gedeelde enumeratieve model checker, waarin een toestandsruimte gaandeweg wordt onderzocht: het begint met een lege graaf, verkent het systeem in de oorspronkelijke toestand, haar opvolgers, de opvolgers van de opvolgers, en zo verder, totdat er een bug gevonden wordt of tot de toestandsruimte is uitgeput. Eigenschappen van oneindige executies worden gaandeweg gecontroleerd met behulp van een cykeldetectie-algoritme door Brim et al. Met behulp van SPINJADI hebben we twee mutual exclusion protocollen en een biologisch model van T-cel activatie tijdens een immuunrespons onderzocht.

In hoofdstuk 5 introduceren we TSCCdc, een efficiënt gedeelde algoritme voor het vinden van terminal sterk verbonden componenten (TSCC’s) in grote grafen. In de biologie komen TSCC’s overeen met toestanden van terminale differentiatie (wanneer een cel stopt met zijn specialisatie), of met stationaire toestanden. TSCCdc is een parallel verdeel-en-heers graafalgoritme: met behulp van bereikbaarheidberekeningen is een graaf te splitsen in vier onafhankelijke subgrafen die niet kunnen worden ‘overgestoken’ via SCC’s, en die zo parallel recursief kunnen worden doorgezocht. Ons algoritme was als enige in staat een realistisch voorbeeldmodel te verwerken: een model van menselijke hematopoëtische (bloed) cellen, met een grootte in de orde van $2^{34}$ knopen en zijden.
Samengevat, de bijdragen van dit proefschrift zijn als volgt. Hoofdstuk 2 introduceert een nieuwe benadering van het modelleren in de biologie, die we gebruiken voor het modelleren van de vulva ontwikkeling in *C. elegans*—dit model bevat twee eerder gepubliceerde, maar niet gemodelleerde hypothesen over het proces. Hoofdstuk 3 bevat een nieuwe schaalbare techniek om de stabilisatie van gelijktijdige systemen aan te tonen. In hoofdstuk 4 ontwerpen en implementeren wij een nieuw raamwerk om het schrijven van gedistribueerde graafalgorithmen te vereenvoudigen; bovendien wordt dit raamwerk gebruikt voor het implementeren van een gedistribueerde verdeel-en-heers enumeratieve model checker. Hoofdstuk 5 introduceert een nieuw efficiënt gedistribueerd algoritme voor het vinden van terminaal sterk verbonden componenten in een grote graaf.