RWTH Aachen

Department of Computer Science
Technical Report

Parallel Algorithms for Verification of Large Systems

Michael Weber

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2006-02

RWTH Aachen · Department of Computer Science · December 2006

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Parallel Algorithms for Verification of Large Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Michael Weber
aus
Düren

Berichter: Prof. Dr. Klaus Indermark
Assoc. Prof. RNDr. Luboš Brim

Tag der mündlichen Prüfung: 24.01.2006

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Abstract

The *model-checking problem* is the question whether a given system model satisfies a property. The property is usually given as formula of a temporal logic, and the system model as labelled transition system. However, the well-known state-space explosion effect is responsible for yielding transition systems of exponential size when compared to their description, and common sequential algorithms often are not capable to solve the model-checking problem with resources available on a single computer.

In this thesis, we develop parallel and, in particular, distributed algorithms which exploit the combined resources of a network of commodity workstations to solve problem instances which are beyond the capabilities of today's sequential algorithms.

Specifically, our algorithms solve the model-checking problem for two important fragments of the μ-calculus which subsume many well-known temporal logics (CTL, LTL, CTL*). We describe our algorithms based on a characterization of the problem at hand in terms of two-player games. The underlying data structure, the *game graph*, is colored according to the player who has a winning strategy from the current game configuration. Finally, the color of the initial configuration tells who is the winner of the game, and thus whether the transition system satisfies the property or not.

Through experimentation, we found that our algorithms scale well, and are able to solve the largest problem instances of the VLTS benchmark suite.

In a second part, we investigate ways to efficiently generate (low-level) transition systems suitable for many verification tools from compact high-level descriptions of the input model. We propose a virtual-machine based approach, which uses an intermediate format to break the translation from high-level to low-level representations of a model into two steps. This well-known compiler technique simplifies the translation and still is very fast in practice.

We show the practicality of our approach through the example of a compiler for the PROMELA modelling language which targets our intermediate language—the virtual-machine's byte-code. With a comparison of benchmarks, we show that our approach is competitive to state-of-the-art tools like SPIN in speed, with additional advantages, like easier reusability, and application as component in distributed model-checking algorithms like the ones we proposed earlier.

Acknowledgements

First and foremost, I sincerely thank my advisor Prof. Dr. Klaus Indermark. From the beginning, he provided me with large degrees of freedom to pursue my research interests. He was a steady source of advice and encouragements, and I enjoyed the many stimulating discussions about scientific, technological and other topics.

I am grateful to Prof. Dr. Luboš Brim for being a member of my thesis committee, and for inviting me to Brno in November 2004, which initiated an interesting and hopefully long-lasting cooperation.

I also thank the graduate college 643 *"Software für Kommunikationssysteme"* for their financial support during the first two years of my research.

Thanks are given to everybody at I2 for a pleasant and friendly research environment, which contributed a lot to this thesis.

I thank my friend and fellow office mate Volker Stolz, for many lively discussions and for suffering through my rants about various research-related and -unrelated topics. Also, Dr. Benedikt Bollig and Dr. Thomas Noll have been great discussion partners through the years.

My special thanks go to my friend and colleague Dr. Martin Leucker, who crossed my way already back during my graduate studies. He sparked my initial interest in Formal Methods, and he has always been a great source of inspiration and advice through the years. I very much enjoyed the many scientific as well as personal discussions.

Last, but certainly not least, my sincerest gratitude goes to Irina for all her love, care, and unfailing support throughout my thesis. I apologize for the long working hours and uneventful weekends during the last year.

Michael Weber
Aachen, November 2005

Contents

1. Thesis **1**
1.1. Objective 2
1.2. Contributions 2
1.3. Overview 4

I. Parallel Model Checking **5**

2. A Classification of Model-checking Algorithms **7**
2.1. Global versus Local Algorithms 7
2.2. Explicit-state versus Symbolic Algorithms 8
2.3. Parallel versus Distributed Algorithms 8
2.4. Related Work 9

3. Parallel Model Checking Games **13**
3.1. Preliminaries 13
3.2. The μ-Calculus 15
3.2.1. Syntax and Semantics 15
3.2.2. Graphs of Formulas 20
3.2.3. Complexity of Model Checking for L^1_μ 25
3.2.4. Model-checking Games for the μ-calculus 27
3.3. Winning L^1_μ-games 32
3.3.1. Sequential Coloring Algorithms 36
3.4. Winning Games for L^1_μ-Formulas in Parallel 46
3.4.1. Distributing the Game Graph 46
3.4.2. Labelling the Game Graph 47
3.4.3. A Family of Parallel Coloring Algorithms 48
3.4.4. Algorithmic Variations and Optimization Issues 52
3.4.5. Calculating Winning strategies 55
3.5. Extensions towards L^2_μ 56
3.5.1. Reducing Alternation Depth 56
3.5.2. Alternation and Game Graphs 57
3.5.3. Coloring Algorithm for L^2_μ 58

4. Implementation and Empirical Results **63**
4.1. The UppDMC Implementation 63
4.2. Practical Experiences 65

II. State Space Generation 73

5. State Space Generation **75**
5.1. Introduction 75
5.2. Status Quo 75
5.3. Contributions 76
5.4. Overview 77

6. Intermediate Formats **79**
6.1. Direct Translation 80
6.2. Using an Intermediate Format 81
6.3. Parallel State Space Generation 83

7. A Virtual Machine-based Approach **85**
7.1. Virtual Machine Specification 85
7.1.1. Machine State 86
7.1.2. Invariants 89
7.1.3. Byte-code Semantics 90
7.1.4. Scheduling 94
7.2. State Space Generation 96
7.3. Use Case: PROMELA 96
7.4. Benchmarks 97
7.5. Evaluation as Intermediate Language 102
7.6. Related Work 104
7.6.1. PROMELA Semantics 104
7.6.2. Virtual Machines 104
7.7. Conclusions 106

8. Conclusions and Future Research **107**

1. Thesis

Model checking [25], originally proposed independently by Emerson and Clarke [40] and Quielle and Sifakis [85], is becoming more and more popular for the verification of complex hardware and software systems. These systems are usually given by means of a formal description that can be transformed into a (labelled) transition system (LTS) which captures the system's essential behavior. In addition, a desired property of the system is usually specified as a formula of a temporal logic. Model-checking algorithms can then answer the question whether the transition system satisfies this property. Numerous case studies have shown that this approach improves the early detection of errors during the design process. An overview is given by Clarke and Wing [26].

However, the well-known *state-space explosion* problem still limits a broader application of model checking. The term refers to the problem that even small system descriptions, when converted to notions digestible by model-checking algorithms, can expand into enormously huge transition systems. This conversion process alone can be very time and space consuming, if done without consideration. In addition, it can lead to unexpected or misleading results if not formalized rigorously.

During the past 20 years, considerable progress in tackling state-space explosion have been achieved. The most prominent examples are *partial-order reduction* [82], *symbolic model checking* [74], and *bounded model checking* [12]. However, typical verification tasks can still last days on a single workstation or are even (practically) undecidable due to memory restrictions (as reported, for example, by Gnesi et al. [46]). Note that even a tenfold reduction in run-time can make the difference between practical feasibility and infeasibility: waiting a single day for a result might be tolerable, while waiting 10 days is very likely too costly.

In contrast, cheap yet powerful parallel computers can be constructed out of *Networks Of Workstations* (*NOW*s). From the outside, a NOW appears as a single parallel computer with high computing power and, even more important, large amounts of memory. A NOW enables parallel programs to utilize its accumulated resources to solve large problem instances, which otherwise would be infeasible.

Various message-passing-interfaces such as MPI [42] or PVM [44] and their corresponding implementations (MPICH[1], LAM[2]) provide application programmers with high-level parallel abstractions and thus allow them to develop portable and efficient parallel programs. In particular, they permit the design of efficient *parallel* model-checking

[1] `http://www-unix.mcs.anl.gov/mpi/mpich/`
[2] `http://www.mpi.nd.edu/lam/`

algorithms which can capitalize on the resources of NOWs. Ideally, these algorithms can then be combined with already well-known techniques for avoiding state-space explosion, thence gaining even higher speedups and further reducing memory requirements.

Another data point which underlines the need for well-performing parallel algorithms can be found in recent developments from the CPU manufacturing industries. Companies like AMD, Intel and Motorola are no longer advertising new products based on raw processor speed, as they are approaching the limits of *Moore's Law*. Instead, they are gradually shifting towards *multi-core processors* which combine more than one processing unit onto a single chip, and inexpensive desktop multi-processor systems, combining several processors into one computer. As a natural extension, several such computers can be bound together into a NOW, building powerful distributed computers from commodity parts.

Sequential model-checking algorithms will only be able to use a fraction of the available computing resources of such machines, and their efficient utilization is demanding the development of new parallel and distributed verification algorithms.

1.1. Objective

With our work, we aim to improve the state-of-the-art in parallel and distributed model-checking algorithms, and to provide algorithms which are able to handle large models by taking full advantage of the upcoming paradigm shift towards parallelism in computer architectures.

To achieve our goals, we investigate along two, mostly orthogonal, axes. First, we address the question how to effectively use parallel resources to solve instances of the model-checking problem in which properties are given as formulas of Kozen's μ-calculus [60], interpreted over labelled transition systems.

In a second step, we propose an approach that bridges the gap between high-level model descriptions suitable for human consumption, and the generation of low-level models (said transition systems) suitable as input to our algorithms. In particular, our approach is fully compatible with the distributed nature of our algorithms.

1.2. Contributions

We support our argument about feasibility and necessity of parallel and distributed algorithms for solving the model-checking problem with the following contributions:

- For model-checking an important sub-logic of the μ-calculus, the *alternation-free* fragment L^1_μ, we first develop a sequential algorithm which admits a scalable distributed version. It is explained in terms of Stirling's model-checking games [91].

- Based on this sequential version, we develop building blocks for a family of distributed model-checking algorithms for the alternation-free fragment L_μ^1, which subsumes the well-known *Computation-Tree Logic* (CTL) [40]. Hence, we get a distributed model-checking algorithm for CTL for free.

  Our algorithms can be tailored to different situations, for example, by trading space efficiency achieved through *on-the-fly* techniques for better worst-case time complexity.

  Our main results on the topic of these algorithms were published in [15]. To the best of our knowledge, this is the first *on-the-fly* parallel algorithm for the alternation-free μ-calculus.

- In a second step, we extend our algorithms to the bigger fragment L_μ^2 of the μ-calculus, thus subsuming *Linear-Time Logic* (LTL) [83] and CTL* [37]. We show that the extension of algorithm is seamless, by reusing our previous algorithms as subroutines.

  Our main result here has been published in [68].

- Finally, to complement our algorithms with a fully formalized, scalable building block for state-space generation, we present a virtual-machine based approach, which is compatible with the on-the-fly characteristics of our algorithms.

  We propose to use a byte-code language as intermediate layer between high-level model descriptions produced by designers, and the low-level transition-system representation of a model suitable for model-checking algorithms. In doing so, we achieve a time and space-efficient solution, which simplifies and opens up possibilities for a number of optimizations.

  To verify our claims, our virtual-machine based approach was employed for the assignment *executable operational semantics* to the well-known specification language PROMELA [54] through a translation to our byte-code language [89]. We were able to plug the virtual machine as component into the DIVINE model-checking library [35], thus deriving a distributed implementation of a PROMELA model-checking tool, called DIVSPIN, on which we reported in [69].

All our building blocks were implemented and experimentally evaluated for feasibility with measurements regarding their behavior in practical situations. Specifically, an implementation of our parallel model-checking algorithms was able to provide all as of yet missing results from the biggest problem instances of the *Very Large Transition Systems* (VLTS) benchmark suite [52].

1.3. Overview

The rest of this paper is divided in two parts. First, we classify model-checking algorithms in Chapter 2 and discuss related work. Then follows the presentation of our parallel algorithms in Chapter 3. We start by introducing the μ-calculus and the game framework. In the main sections 3.4 and 3.5 of this chapter we present our algorithms which solve the model-checking problem for the fragments L_μ^1 and L_μ^2 in parallel. We conclude the model-checking part in Chapter 4 with measurements and experimental results results.

In the second part, we investigate a virtual machine-based approach to state-space generation. In Chapter 5 we highlight common problems of the translation from models suitable for designers and verification engineers, to low-level representations needed for most model-checking algorithms, ours included. In Chapter 6 we review the benefits of introducing an intermediate step into the translation, and recollect design criterions for intermediate formats. We propose our own intermediate format in form of a virtual-machine based approach in Chapter 7. We formalize a specific virtual-machine instance, and evaluate the practicality of this approach with the help of experiments.

Finally, a brief summary is given in Chapter 8, along with conclusions and an outlook on future work.

Part I.

Parallel Model Checking

2. A Classification of Model-checking Algorithms

The input for the class of algorithms discussed in this thesis is a specification given in a temporal logic (in our case, the μ-calculus), and a system model given in form of a labelled transition system.

Before we develop and present our model-checking algorithms, we briefly review the choices we can take within the design space.

2.1. Global versus Local Algorithms

Model-checking algorithms, regardless whether sequential or parallel, can be broadly classified into *global* and *local algorithms*. Global algorithms require that the transition system is constructed completely, while local algorithms compute *on-the-fly* only those parts of a transition system which directly affect the result.

As an immediate consequence, local (or on-the-fly) algorithms are advantageous if merely a small fraction of the input transition system is actually processed: storage space can be reduced, and also the run-time of algorithms, as the upfront construction of the whole transition system is wasteful if most of it turns out to be irrelevant for the problem at hand. Still, in the worst case even local algorithms might be forced to construct the whole transition system in order to give a result. A prominent example is the check for absence of deadlocks in a model, which requires to check that no reachable state of the underlying transition system is without outgoing edges. Unless the transition system indeed contains such a state (in which case a negative answer can be given as soon as it is found), the whole transition system must be checked.

Worse yet, even the potential savings of a local algorithm usually come at price, namely that their worst-case theoretical bounds are less good when compared to their global counterparts. Global algorithms are less constrained by the structure of the transition system. They have access to all of it from the beginning, and as such can compute solutions inductively in a bottom-up manner, for example. In contrast, local algorithms must operate with a top-down view only which can cause work to be redone.

It is a generally accepted view that global algorithms are to be preferred if it is clear from the beginning that the properties to be checked require large parts of the transition system to be constructed. Another situation in favor of global approaches are later stages of the design process when models mostly remain unmodified. It might then

make sense to store its transition system, and reuse it for checking several properties. Full storage also opens up options for intermediate optimization or compression of a transition system to reduce the amount of work done by a subsequent run of a model-checking algorithm.

On-the-fly algorithms have been proven superior in early design phases when models still contain many errors and hence are prone to change often.

In our work, we first develop a global algorithm and then adapt it towards an on-the-fly behavior. However, our interest is clearly biased towards this latter class, as these algorithms often find errors faster and with less storage resources, and are able to provide solutions on cases where global algorithms exhaust all available resources and fail.

2.2. Explicit-state versus Symbolic Algorithms

The second decision to be taken in the model-checking design space is due to the dichotomy between *explicit-state algorithms* and *symbolic algorithms*.

In explicit-state approaches, states of a transition system are handled discretely, and each state is stored "as-is", which is why these algorithms are sometimes called *enumerative*.

Symbolic algorithms, in contrast, operate on sets of states rather than single states, usually in form of a global fixpoint iteration. The set of already visited states and the edge relation of a transition system are often encoded as *binary decision diagrams* (BDDs) or variants thereof. Depending on the structure of the transition system, these encodings can be very compact, thus allowing excessively large models to be stored and processed.

For example, BDDs have been used successfully in hardware verification. For models of software they have not fared so well. Hu et al. state that *"BDD-based algorithms [...] appear generally unable to handle designs much more complex than dining philosophers or rings of mutual exclusion elements"* [56]. Furthermore, they report that approaches with explicit state representation as well as with BDDs both have application domains in which they outperform their counterpart.

Here, we concentrate on explicit-state algorithms because we are in particular interested in models stemming from communication protocols and concurrent systems.

2.3. Parallel versus Distributed Algorithms

A *parallel algorithm* can take advantage of more than one processor by dividing the work to be done into pieces, which can then be handled independently. Eventually, partial results are collected and incorporated into the final solution.

distributed algorithms are a subclass of parallel algorithms which have the additional constraint that their processors do not share main memory, but are loosely connected via a network. The advantage of such setups is that they are are inexpensive and can be composed from standard components, whereas large shared-memory computers traditionally consisted of proprietary special-purpose hardware. However, because a network connection is orders of magnitude slower than a direct memory access, algorithms specially tailored to this situation need to be devised in order to be competitive.

Our algorithms are expressed as *message-passing algorithms*, thus targeting mainly distributed environments. In addition, the use of standardized libraries allows us to run our algorithms even on shared-memory architectures, with little overhead.

2.4. Related Work

Despite more than twenty years of research in the area of model checking algorithms, until recently not much effort has been spent on their *parallelization*. However, this topic becomes increasingly attractive, as the architecture of computers changes towards a more parallel (or even distributed) design.

Stornetta [92] and Basonov [8] present parallelized data structures which exploit additional computers within a network as a substitute for external storage.

The algorithms described by Narayan et al. [77] and Cabodi et al. [22] divide the underlying problem into several tasks. However, they are designed in a way that only a single computer can be employed to sequentially handle one task at a time. Stern and Dill [90] show how to carry out a parallel reachability analysis. The distribution of the underlying structure is similar to the one presented here. Their algorithm is limited as it only allows reachability checking, but is not appropriate for general model checking of temporal logic formulas.

Heyman et al. developed a parallel reachability analysis algorithm for BDDs [50]. They argue that many safety properties can be formulated as a reachability problem. In this way, their algorithm allows checking safety formulas. However, *liveness* properties which can be expressed within L_μ^1 are not supported.

Grumberg et al. [49] introduced a symbolic parallel algorithm for the full μ-calculus. However, it is global and thus requires full construction of the transition system.

The algorithms we present are based on a characterization of the model checking problem for fragments of the μ-calculus in terms of two-person games due to Stirling [91]. Strictly speaking, we present a parallel algorithm for coloring *game graphs* corresponding to the underlying model checking problem. This coloring answers the model checking problem. Furthermore, we explain that our algorithm can be extended to compute winning strategies without further costs. A strategy may be employed by the user of a verification tool for debugging the underlying system interactively [91].

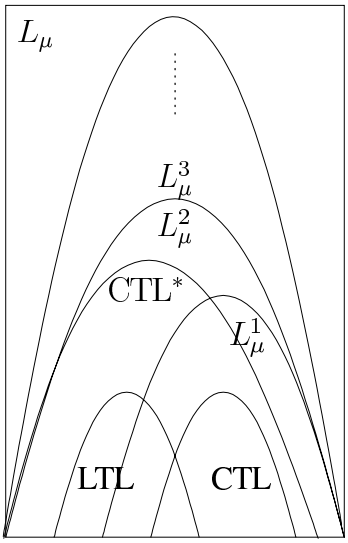


Figure 2.1.: Expressiveness hierarchy of temporal logics

A different characterization of the model checking problem can be given in terms of one-letter-simple-weak-alternating-Büchi automata (1SWABA) [61]. However, these are related to games in a straightforward manner [65]. Hence, our algorithm can also be understood as a parallel procedure for checking the emptiness of this kind of automata.

On the same line, we can view the model-checking problem as solving a *boolean equation system* (BES) [72]. Very recently, Joubert and Mateescu [59] described a distributed local resolution algorithm in terms of alternation-free BESs. They credit us in their section about related work for the only other distributed resolution algorithm they are aware of. Their algorithm is very similar to ours, and provides just a different way to express the same solution (from private communication with the authors).

The first variation of our parallel algorithm is similar to a solution of the model checking problem described by Kupferman et al. [61]. However, their proposed algorithm employs the detection of cycles which is unlikely to be parallelized in a simple way. Our key observation is that we can omit this step by exploiting structural information of the underlying graph. Furthermore, we present a second version of our algorithm which provides extended on-the-fly behavior for a cheap price.

We then turn our focus towards the richer fragment of the μ-calculus allowing one alternation. It is of practical importance since it subsumes Linear Temporal Logic, LTL [83], as well as CTL* [37], which follows by (unpublished) results from Wolper, as well as Emerson and Lei [38], and was shown in a direct manner by Dam [31]. We develop a parallel model checking algorithm for μ-calculus formulas up to alternation depth 2 (Figure 2.1).

Our reduction allows a promising approach to check formulas from LTL, CTL*, and

L^2_μ within the same framework. However, it has to compete with specialized algorithms for the logics mentioned. For instance, Brim, Barnat, et al. proposed several algorithms tailored to distributed LTL model-checking [5, 20, 6, 7].

3. Parallel Model Checking Games

All our algorithms have in common that they operate on graphs in various flavors as fundamental structures. Before we formally define the the μ-calculus and present our algorithms, we first fix some graph theoretic notions and notations which are used throughout the following sections.

3.1. Preliminaries

Definition 3.1.1 (Graph)
A *directed graph* (*digraph*) $\mathcal{G}$ is a structure $\mathcal{G} = (Q, \rightarrow)$ where Q is an arbitrary set and $\rightarrow \subseteq Q \times Q$. We call the elements of Q *nodes* of $\mathcal{G}$ and the elements of $\rightarrow$ we call *edges* of $\mathcal{G}$.

Definition 3.1.2 (Path, Cycle)
For $q, q' \in Q$, a *path* from q to q' is a sequence of nodes $q_0, \ldots, q_n \in Q$ such that $q = q_0 \rightarrow \ldots \rightarrow q_n = q'$. A *cycle* in $\mathcal{G}$ is a sequence of nodes $q_0, \ldots, q_n$ such that $q_0 \rightarrow \ldots \rightarrow q_n \rightarrow q_0$. We say a node $q \in Q$ *is contained* or *reached in* a cycle iff there is a cycle $q_0, \ldots, q_n$ in $\mathcal{G}$ and $q = q_i$ for some $i \in \{0, \ldots, n\}$.

Definition 3.1.3 (Connected Graph, DAG, Tree)
A graph $\mathcal{G} = (Q, \rightarrow)$ is *connected*, iff there exists a node $q \in Q$ such that there exist paths from q to any other node, viz $\exists q \in Q : \ \forall q' \in Q : \ q = q' \vee q = q_0 \rightarrow q_1 \rightarrow \cdots \rightarrow q_n = q'$.

A digraph $\mathcal{G}$ that does not contain any cycles is classified as *directed acyclic graph*, or DAG for short.

A *tree* $\mathcal{T} = (Q, \rightarrow)$ is a connected DAG such that $\forall q, q', q'' \in Q : q \rightarrow q'' \ \wedge \ q' \rightarrow q'' \Rightarrow q = q'$.

Definition 3.1.4 (Component)
A *component* of a graph $\mathcal{G}$ is a subgraph $\mathcal{G}' \subseteq \mathcal{G}$ induced by a set of nodes Q', that is $\mathcal{G}' = (Q', \rightarrow')$, $Q' \subseteq Q$, and $\rightarrow' = \rightarrow \cap (Q' \times Q')$. If the context is unambiguous, we also call the inducing set Q' a component.

A *(strongly) connected component* (SCC) is a component $\mathcal{G}' = (Q', \rightarrow')$ of $\mathcal{G}$ such that for all $q, q' \in Q'$ there is a path from q to q' in $\mathcal{G}'$. A (connected) component and a cycle are called *non-trivial* if they contain a least two nodes.

We consider a connected component $\mathcal{G}'$ *maximal* with regard to $\mathcal{G}$, iff there exists no other connected component $\mathcal{G}'' \subseteq \mathcal{G}$ such that $\mathcal{G}' \subsetneq \mathcal{G}''$.

Definition 3.1.5 (Bridge)
Let $\mathcal{G}' = (Q', \rightarrow')$ and $\mathcal{G}'' = (Q'', \rightarrow'')$ be two disjoint components of $\mathcal{G}$, that is $\mathcal{G}'$ and $\mathcal{G}''$ are components of $\mathcal{G}$ such that $Q' \cap Q'' = \varnothing$. Furthermore, assume that there is no edge from a node in Q'' to a node in Q', thus $\rightarrow \cap (Q'' \times Q') = \varnothing$. Then we call every edge from a node $q' \in Q'$ to a node $q'' \in Q''$ ($q' \rightarrow q''$) a *bridge*.

In the following sections, we consider node-labelled graphs $\mathcal{G} = (Q, \rightarrow, \lambda)$, where $(Q, \rightarrow)$ is a graph and λ is a labelling function from states to some domain. In particular, we deal with graphs where nodes are labelled by formulas.

Definition 3.1.6 (Partially Ordered Set)
Let S be a finite set. The binary relation $\leq$ is a *partial order* on S, iff $\leq$ is *reflexive*, *antisymmetric*, and *transitive*, that is:

- $x \leq x$, for all $x \in S$ (reflexivity)
- $x \leq y$ and $y \leq x$ implies $x = y$, for all $x, y \in S$ (antisymmetry)
- $x \leq y$ and $y \leq z$ implies $x \leq z$, for all $x, y, z \in S$ (transitivity)

The pair $\mathcal{S} = (S, \leq)$ is called a *partially ordered set*, or short *poset*.

Definition 3.1.7 (Cover Relation)
We define the *interval* $[x, y]$ in poset $\mathcal{S} = (S, \leq)$ as the set of all $z \in S$, such that $x \leq z$ and $z \leq y$.
If $[x, y] = \{x, y\}$, then we call y a *cover* of x, and $x \lessdot y$ the corresponding *cover relation*.

A well-known graphical representation of a cover relation is its Hasse diagram:

Definition 3.1.8 (Hasse Diagram)
A *Hasse diagram* of poset $\mathcal{S} = (S, \leq)$ is a directed graph $\mathcal{G} = (S, \rightarrow)$. We draw an edge $x \rightarrow y$ iff $x \lessdot y$.

Usually, Hasse diagrams are drawn with an implied upward orientation and hence edges are drawn without arrows.

Definition 3.1.9 (Tree Order)
A *tree order* is a structure $(Q, \leq)$ such that $\leq$ is a partial order on Q and its Hasse diagram is a tree. More precisely, we call $\leq$ a *tree order* iff

- there is a unique $q \in Q$ such that $q \leq q'$ for all $q' \in Q$, and
- for all $q, q', q'' \in Q$, $q \lessdot q''$ and $q' \lessdot q''$ implies $q = q'$.

Notions of parents and children for elements of Q with respect to $\leq$ correspond to the usual ones for elements of Q with respect to $\lessdot$.

3.2. The μ-Calculus

A famous logic for expressing specifications is Kozen's μ-calculus [60], a temporal logic offering boolean combination of formulas and, especially, labelled *next*-state, minimal, and maximal fixpoint quantifiers. It is the most expressive logic adequate for transition systems representing concurrent systems: It is expressively complete with respect to the fragment of second order logic consisting of the formulas not distinguishing *bisimilar* transition systems [57]. One of the most important unanswered questions about the μ-calculus is whether there is a polynomial time model checking algorithm improving the known bounds, viz NP and Co-NP [39].

For practical applications, however, it suffices to restrict the μ-calculus in order to gain tractable model checking procedures. The alternation-free fragment, denoted by L_μ^1, prohibits the nesting of minimal and maximal fixpoint operators. It allows the formulation of many *safety* as well as *liveness* properties. While this fragment is already important on its own, it subsumes *Computation-Tree Logic*, CTL [40], which is employed in many practical verification tools. It can be shown that the model checking problem for this fragment is linear in the length of the formula as well as the size of the underlying transition system, and several sequential model checking procedures are given in the literature [29, 1, 61, 11].

In this section, we recall the syntax and semantics of the modal μ-calculus.

3.2.1. Syntax and Semantics

Let Var be a set of fixpoint variables, Prop be a set of propositional variables, and Σ be a finite set of actions. We require the fixpoint variables to be distinct from the propositional variables ($\mathit{Var} \cap \mathit{Prop} = \varnothing$).

Definition 3.2.1 (μ-Calculus Syntax)
Formulas of the propositional modal μ-calculus over Var and Σ in positive form as introduced by Kozen [60] are defined through the following BNF grammar:

$$\varphi ::= \texttt{false} \mid \texttt{true} \mid X \mid \neg p \mid p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [K]\varphi \mid \langle K \rangle \varphi \mid \nu X.\varphi \mid \mu X.\varphi$$

where $X \in \mathit{Var}$, $p \in \mathit{Prop}$, and K ranges over subsets of actions Σ. We denote as L_μ the language generated by φ.

Besides boolean constants `false` and `true`, fixpoint variables and (possibly negated) propositional variables, we allow the usual boolean connectives, as well as *modalities* ($\langle\cdot\rangle$ and $[\cdot]$), and maximal and minimal *fixpoint operators* (ν and μ). Whenever the specific type of a fixpoint operator does not matter, we use σ for either μ or ν, and $\overline{\sigma}$ for its dual, that is, if $\sigma = \mu$ then $\overline{\sigma} = \nu$, and vice versa. Similarly, we write $[\![K]\!]$ for $\langle K \rangle$ or $[K]$, $\star$ for $\vee$ or $\wedge$, and $\#\varphi$ for $\sigma X.\, \varphi$ or $[\![K]\!]\varphi$.

We follow Stirling's lead [91] and allow sets of actions instead of single actions appearing in modalities. For convenience, we allow some abbreviations:

$$\begin{aligned} [\![a]\!]\varphi \quad &\text{instead of} \quad [\![\{a\}]\!]\varphi \\ [\![-a]\!]\varphi \quad &\text{instead of} \quad [\![\Sigma\backslash\{a\}]\!]\varphi \\ [\![-K]\!]\varphi \quad &\text{instead of} \quad [\![\Sigma\backslash K]\!]\varphi \\ [\![-]\!]\varphi \quad &\text{instead of} \quad [\![\Sigma]\!]\varphi \end{aligned}$$

For μ-calculus formulas, we introduce the notion of *subformulas*, *free*, and *bound* variables as usual.

Definition 3.2.2 (Subformula)
The *subformulas* $\mathrm{Sub}(\varphi) \subseteq L_\mu$ of a formula φ are inductively defined as:

$$\begin{aligned} \mathrm{Sub}(\varphi) &= \{\varphi\} && \text{for } \varphi \in \{\mathtt{false}, \mathtt{true}, X, p, \neg p\} \\ \mathrm{Sub}(\varphi \star \psi) &= \{\varphi \star \psi\} \cup \mathrm{Sub}(\varphi) \cup \mathrm{Sub}(\psi) \\ \mathrm{Sub}([\![K]\!]\varphi) &= \{[\![K]\!]\varphi\} \cup \mathrm{Sub}(\varphi) \\ \mathrm{Sub}(\sigma X.\, \varphi) &= \{\sigma X.\, \varphi\} \cup \mathrm{Sub}(\varphi) \end{aligned}$$

Definition 3.2.3 (Free and Bound Variables)
We define the set of *free variables* $\mathrm{FV}(\varphi) \subseteq \mathit{Var}$ of a formula $\varphi \in L_\mu$ inductively:

$$\begin{aligned} \mathrm{FV}(\mathtt{false}) &= \varnothing = \mathrm{FV}(\mathtt{true}) \\ \mathrm{FV}(\neg p) &= \varnothing = \mathrm{FV}(p) \\ \mathrm{FV}(X) &= \{X\} \\ \mathrm{FV}(\varphi \star \psi) &= \mathrm{FV}(\varphi) \cup \mathrm{FV}(\psi) \\ \mathrm{FV}([\![K]\!]\varphi) &= \mathrm{FV}(\varphi) \\ \mathrm{FV}(\sigma X.\, \varphi) &= \mathrm{FV}(\varphi)\backslash\{X\} \end{aligned}$$

The set of *bound variables* $\mathrm{BV}(\varphi) \subseteq \mathit{Var}$ is inductively defined by:

$$\begin{aligned} \mathrm{BV}(\mathtt{false}) &= \varnothing = \mathrm{BV}(\mathtt{true}) \\ \mathrm{BV}(\neg p) &= \varnothing = \mathrm{BV}(p) = \mathrm{BV}(X) \\ \mathrm{BV}(\varphi \star \psi) &= \mathrm{BV}(\varphi) \cup \mathrm{BV}(\psi) \\ \mathrm{BV}([\![K]\!]\varphi) &= \mathrm{BV}(\varphi) \\ \mathrm{BV}(\sigma X.\, \varphi) &= \mathrm{BV}(\varphi) \cup \{X\} \end{aligned}$$

Consequently, we call σX a *binder* of variable X. We call X a *ν-variable* or to be of *type ν*, if it is bound by a ν-binder, and μ-variable or of type μ, otherwise.

A formula φ is a *sentence* iff φ contains no free fixpoint variables ($\mathrm{FV}(\varphi) = \varnothing$). Furthermore, we say that φ is a μ-formula iff $\varphi = \mu X.\psi$ for appropriate X and ψ. ν-formulas are introduced correspondingly. From now on, we require all formulas to be sentences.

To simplify the cases to consider further on, we restrict formulas with a syntactic argument. We consider a formula φ *normal* if all occurrences of binders σX in φ bind a distinct variable X. Through renaming, every formula can easily be converted into an equivalent normal formula. For example, $(\mu X.X) \vee (\mu X.X)$ is not normal whereas $(\mu X.X) \vee (\mu Y.Y)$ is normal.

Remark 3.2.4
If a formula φ is normal, every bound variable $X \in \mathrm{BV}(\varphi)$ of φ identifies a unique subformula $\sigma X.\Phi_X \in \mathrm{Sub}(\varphi)$ where $X \in \mathrm{FV}(\Phi_X)$ is a free variable of Φ_X. In the following, we assume all formulas to be normal.

So far, we have been considering purely syntactic features of L_μ. The semantics of formulas is usually defined by interpreting them on finite labelled transition systems, which we will now introduce.

Definition 3.2.5 (Labelled Transition System)
We define $\mathcal{T} = (S, \rightarrow, \Sigma, s_0, P)$ as *labelled transition system* where S is a finite set of states, Σ a finite set of actions, and $\rightarrow \subseteq S \times \Sigma \times S$ denotes the transitions. As usual, we write $s \xrightarrow{a} t$ instead of $(s, a, t) \in \rightarrow$. Furthermore, let $s_0 \in S$ be the *initial state* of the transition system and let $P : \mathit{Prop} \rightarrow 2^S$ denote a function yielding the set of states in which a given proposition holds.

A valuation V maps a fixpoint variable X to a set of states $V(X) \subseteq S$. Let $V[X/E]$, $E \subseteq S$, be the valuation which is the same as V except for X where $V(X) = E$.

Definition 3.2.6 (μ-Calculus Semantics)
Given a labelled transition system $\mathcal{T} = (S, T, \Sigma, s_0, P)$, a formula φ over Var, Prop, and Σ, and a valuation V, the semantics of φ is a set of states $[\![\varphi]\!]^{\mathcal{T}}_V \subseteq S$ in which φ

holds, inductively defined as follows:

$$
\begin{aligned}
[\![\texttt{true}]\!]^{\mathcal{T}}_V &:= S \\
[\![\texttt{false}]\!]^{\mathcal{T}}_V &:= \varnothing \\
[\![X]\!]^{\mathcal{T}}_V &:= V(X) \\
[\![p]\!]^{\mathcal{T}}_V &:= P(p) \\
[\![\neg p]\!]^{\mathcal{T}}_V &:= S \setminus [\![p]\!]^{\mathcal{T}}_V \\
[\![\varphi_1 \vee \varphi_2]\!]^{\mathcal{T}}_V &:= [\![\varphi_1]\!]^{\mathcal{T}}_V \cup [\![\varphi_2]\!]^{\mathcal{T}}_V \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathcal{T}}_V &:= [\![\varphi_1]\!]^{\mathcal{T}}_V \cap [\![\varphi_2]\!]^{\mathcal{T}}_V \\
[\![[K]\varphi]\!]^{\mathcal{T}}_V &:= \left\{ s \in S \mid \forall s \xrightarrow{a} t :\ a \in K \Rightarrow t \in [\![\varphi]\!]^{\mathcal{T}}_V \right\} \\
[\![\langle K \rangle \varphi]\!]^{\mathcal{T}}_V &:= \left\{ s \in S \mid \exists s \xrightarrow{a} t :\ a \in K \wedge t \in [\![\varphi]\!]^{\mathcal{T}}_V \right\} \\
[\![\mu X.\, \varphi]\!]^{\mathcal{T}}_V &:= \bigcap \left\{ E \subseteq S \mid E = [\![\varphi]\!]^{\mathcal{T}}_{V[X/E]} \right\} \\
[\![\nu X.\, \varphi]\!]^{\mathcal{T}}_V &:= \bigcup \left\{ E \subseteq S \mid E = [\![\varphi]\!]^{\mathcal{T}}_{V[X/E]} \right\}
\end{aligned}
$$

If the transition system is clear from the context we leave it out and write $[\![\varphi]\!]_V$. Also, if φ is a sentence then the set $[\![\varphi]\!]_V$ does not depend on V and in this case we may write $[\![\varphi]\!]$ to denote the semantics of φ. We write $\mathcal{T}, s \models \varphi$ and say that sentence φ is satisfied in state s of a transition system $\mathcal{T}$ if $s \in [\![\varphi]\!]^{\mathcal{T}}$.

Throughout the rest of our work we use identifiers like $\varphi, \psi, \ldots$ for formulas, Φ_X for formulas containing a free fixpoint variable X, $s, t, \ldots$ for states, and $a, b, \ldots$ for actions of the transition system under consideration. K denotes a set of actions.

Fixpoint Iteration. Fixpoints can be computed by standard iteration. The domain of the function is the finite complete partial order[1] (cpo) 2^S in which each chain has length at most $|S|$. Hence, to find the fixpoint we need at most $|S|$ iterations.

For some formula $\sigma X.\, \Phi_X$ we can characterize its *iteration semantics* as follows:

$$
\begin{aligned}
\mathcal{F}^0 &:= \begin{cases} \varnothing & \text{if } \sigma = \mu \\ S & \text{if } \sigma = \nu \end{cases} \\
\mathcal{F}^{i+1} &:= [\![\Phi_X]\!]_{V[X/\mathcal{F}^i]}
\end{aligned}
$$

We start with the empty set or the set of all states, depending on the fixpoint type. In each iteration, we update the valuation V for X with the results from the previous iteration.

[1] As usual, we use notation $2^S := \{S' \mid S' \subseteq S\}$ to denote the powerset of a set S.

The iteration stops when $\mathcal{F}^i = \mathcal{F}^{i+1}$. Hence, in iteration i the result $\mathcal{F}^i$ coincides with the one from the semantics defined above: $\mathcal{F}^i = [\![\sigma X.\, \Phi_X]\!]$.

We say variable Y is *subsumed* by variable X (notation $Y \sqsubseteq X$) if $\sigma Y.\, \Phi_Y$ is a subformula of Φ_X: $\sigma Y.\, \Phi_Y \in \mathrm{Sub}(\Phi_X)$. Note that if X subsumes Y then each computation of $\mathcal{F}^i = [\![\Phi_X]\!]_V$ can potentially lead to a nested fixpoint computation of $[\![\sigma Y.\, \Phi_Y]\!]_{V[X/\mathcal{F}^i]}$. However, if X does not occur free in Φ_Y ($X \notin \mathrm{FV}(\Phi_Y)$) then the nested fixpoint needs to be computed only once since its value does not depend on the current value $\mathcal{F}^i$ of X. Otherwise the nested fixpoint must be recalculated each time the value of X is updated. We write variable Y *depends on* X (notation $Y \prec X$) if $Y \sqsubseteq X$ and $X \in \mathrm{FV}(\Phi_Y)$.

The *alternation depth* $\mathrm{ad}(\varphi)$ of a formula φ is the length of the longest chain $X_1 \prec X_2 \prec \cdots \prec X_d$ of variables occurring in φ such that X_i and X_{i+1} are of different type for each i.

Alternation depth can be also defined for a single variable. In that case we look at chains of alternating variables which end up in the given one. Formally, the alternation depth $\mathrm{ad}(X)$ of a variable $X \in \mathrm{BV}(\varphi)$ is the length of the longest chain $X_1 \prec X_2 \prec \cdots \prec X_d = X$ of variables of alternating types. Straightforwardly, $\mathrm{ad}(\varphi) = \max\{\mathrm{ad}(X) \mid X \in \mathrm{BV}(\varphi)\}$.

Note that alternation depth is a syntactic criterion, and thus can also be defined inductively on the formula structure [95].

Definition 3.2.7 (Alternation Depth)
The *alternation depth* $\mathrm{ad}(\varphi)$ of a formula $\varphi \in L_\mu$ is defined as

$$
\begin{aligned}
\mathrm{ad}(\texttt{false}) &= 1 = \mathrm{ad}(\texttt{true}) \\
\mathrm{ad}(\neg p) &= 1 = \mathrm{ad}(p) \\
\mathrm{ad}(X) &= 1 \\
\mathrm{ad}(\varphi_1 \star \varphi_2) &= \max(\mathrm{ad}(\varphi_1), \mathrm{ad}(\varphi_2)) \\
\mathrm{ad}([\![K]\!]\varphi) &= \mathrm{ad}(\varphi) \\
\mathrm{ad}(\sigma X.\, \varphi) &= \mathrm{ad}(\varphi) + \begin{cases} 1 & \text{if } \exists \psi \in \mathrm{Sub}(\varphi):\ \psi = \overline{\sigma} Y.\, \Psi_Y \text{ and } X \in \mathrm{FV}(\Psi_Y) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that a formula φ is called *alternation-free* if $\mathrm{ad}(\varphi) = 1$.

Definition 3.2.8 (Fragments of L_μ)
We denote by L_μ^n the set of all μ-formulas up to alternation depth n and by L_μ the set of all μ-calculus formulas.

The fragment L_μ^1 is called the *alternation-free* fragment of L_μ. It is well-known that the alternation-depth hierarchy of μ-calculus formulas is strict [19]. In particular, alternation depth 2 is needed to capture expressive power of LTL and CTL* [31, 38] (Figure 2.1).

3.2.2. Graphs of Formulas

Essential for our further development is a formula's graph representation. To simplify the definition, we first introduce the *tree representation* of a formula, also called parse tree. Let φ be a formula. The *occurrence set* of φ denoted by $Occ(\varphi)$ is inductively defined as

$$\begin{aligned} \epsilon &\in Occ(\varphi) \\ i\pi &\in Occ(\varphi) && \text{if } i \in \{1,2\}, \varphi = \varphi_1 \star \varphi_2 \text{ and } \pi \in Occ(\varphi_i) \\ 1\pi &\in Occ(\varphi) && \text{if } \varphi = \#\varphi' \text{ and } \pi \in Occ(\varphi') \end{aligned}$$

Let $\varphi|_\pi$ denote the subformula of φ at position π, that is

$$\begin{aligned} \varphi|_\epsilon &:= \varphi \\ \varphi|_{i\pi} &:= \varphi_i|_\pi && \text{where } i \in \{1,2\} \text{ and } \varphi = \varphi_1 \star \varphi_2 \\ \varphi|_{1\pi} &:= \varphi'|_\pi && \text{where } \varphi = \#\varphi' \end{aligned}$$

We assign to every φ a $\mathrm{Sub}(\varphi)$-labelled tree with nodes $Occ(\varphi)$ and edge relation $\rightarrow$.

Definition 3.2.9 (Tree Representation of L_μ-Formulas)
The *tree representation* of a formula $\varphi \in L_\mu$ is defined as $\mathcal{T}(\varphi) = (Occ(\varphi), \rightarrow, \lambda)$, with edge relation

$$\rightarrow := \{(\pi, i\pi) \mid \pi, i\pi \in Occ(\varphi), i \in \mathbb{N}, \pi \in \mathbb{N}^*\}$$

and labels $\lambda(\pi) = \varphi|_\pi$.

We are now ready to define the *graph representation* of a formula φ. Intuitively, the graph of a formula is its canonical tree representation enriched with edges from fixpoint variables back to the fixpoint formula it identifies.

Definition 3.2.10 (Graph Representation of L_μ-Formulas)
Let $\varphi \in L_\mu$ be a formula and $\mathcal{T}(\varphi) = (Occ(\varphi), \rightarrow, \lambda)$ be its tree representation. The *graph of* φ denoted by $\mathcal{G}(\varphi)$ is the tuple $(Occ(\varphi), \rightarrow', \lambda)$ where

$$\rightarrow' = \rightarrow \cup \{(\pi, \pi') \mid \lambda(\pi) = X \text{ and } \lambda(\pi') = \sigma X.\, \Phi_X \text{ for } X \in \mathrm{BV}(\varphi)\}$$

Because we require φ to be a sentence in normal form, there is for all positions π with $\lambda(\pi) = X$ exactly one matching position π' with $\lambda(\pi') = \sigma X.\, \Phi_X$. However, in general there can be several such positions π.

The graph of the formula $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$ is shown in Figure 3.1.

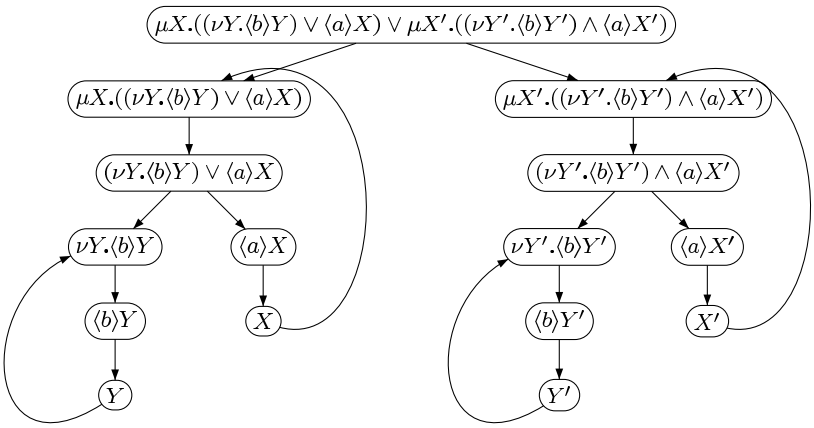
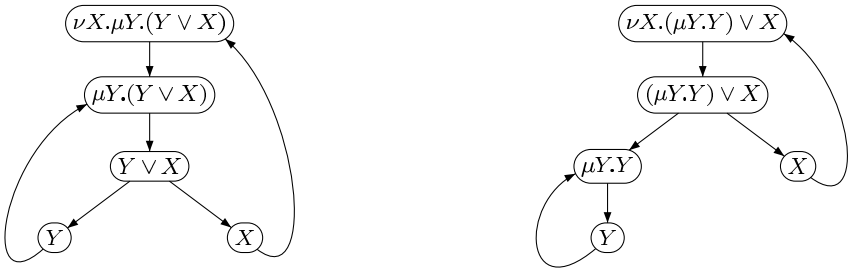


Figure 3.1.: The graph for $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$.



(a) one alternation

(b) alternation-free

Figure 3.2.: Alternation ($\nu X.(\mu Y.Y \vee X)$) vs. alternation-free ($\nu X.(\mu Y.Y) \vee X$)

Recall that the *alternation-free fragment* of the μ-calculus is the sub-logic of the μ-calculus where no subformula ψ of a formula φ contains both a free variable X bound by some μX as well as a free variable Y bound by a νY. In terms of the graph representation, a formula φ is alternation-free iff $\mathcal{G}(\varphi)$ contains no cycle with both a ν-variable and a μ-variable. Figure 3.2(a) shows the graph of an alternating formula which has a cycle containing X as well as Y. Conversely, Figure 3.2(b) shows the graph of an alternation-free formula which has two maximal strongly connected components, one on which X occurs, a second containing Y.

For the next sections, we will restrict ourselves to the alternation-free fragment of the μ-calculus L^1_μ. An extension for the case of one alternation is presented in Section 3.5.

A straight-forward, but essential observation is that the graph of a formula can naturally be decomposed into maximal strongly connected components and trees (see Figure 3.3). We will now formalize this decomposition.

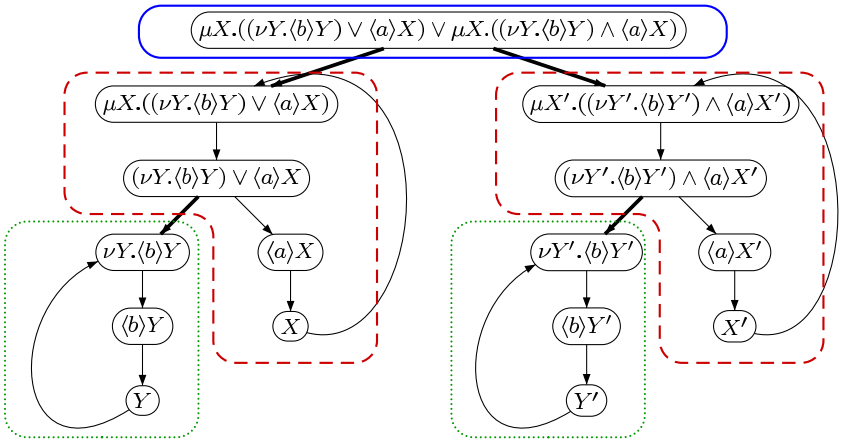


Figure 3.3.: Partition of the formula graph for $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$. Bridge edges are highlighted.

Lemma 3.2.11
Let $\varphi \in L^1_\mu$ be an alternation-free μ-calculus formula and $\mathcal{G}(\varphi) = (Q, \to, \lambda)$ be its graph representation. Then there exists a set of components $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ such that the following holds:

1. The set $\mathcal{Q}$ is a partition of Q, that is $Q = \bigcup \mathcal{Q}$ and for all $i, j \in \{1, \ldots, m\}$ with $i \neq j$, Q_i and Q_j are disjoint $Q_i \cap Q_j = \varnothing$.

2. Every subgraph induced by $Q_i \in \mathcal{Q}$ is either

   a. a non-trivial maximal strongly connected component in which only either μ-formulas or ν-formulas occur, or

   b. a maximal tree with regard to $\mathcal{Q}$, that is for all components $Q' \in \mathcal{Q}$ if $Q' \cup Q_i$ induces a tree then $Q' = Q_i$.

3. There exists a tree order $\leq$ on $\mathcal{Q}$ such that for every edge $q \to q'$ with $q \in Q_i$ and $q' \in Q_j$, we have $Q_i \leq Q_j$. More specifically, edges from a node in Q_i lead to configurations in either the same Q_i or a child Q_j. Without loss of generality, we may assume that $Q_i \leq Q_j$ implies $i \leq j$, so that smaller components with regard to $\leq$ have lower indexes.

Proof. First, we consider nodes of maximal non-trivial strongly connected components. Due to the construction of $\mathcal{G}(\varphi)$ from $\mathcal{T}(\varphi)$, a cycle can occur only through edges from a fixpoint variable to its associated binder. Further, alternation-freeness guarantees that not both a ν-variable and a μ-variable is reached on a cycle.

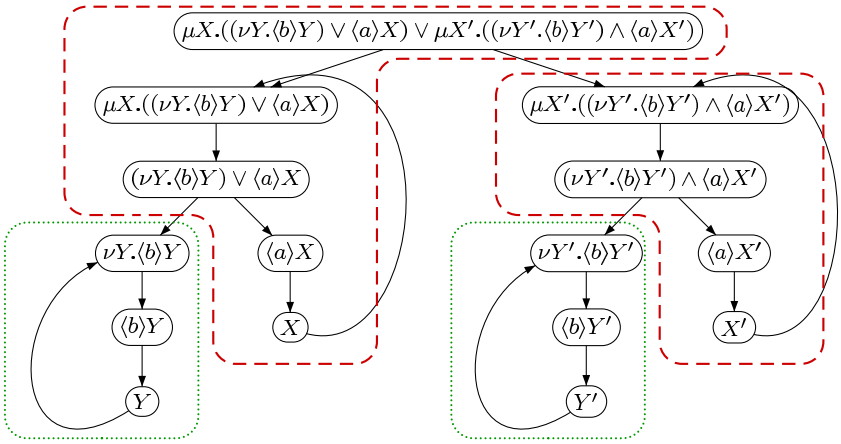


Figure 3.4.: A coarser partition of the graph for the formula $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$. The top-most component was collated with its left child component.

Again, due to the construction of $\mathcal{G}(\varphi)$ those nodes not belonging to a cycle are indeed part of trees. As a formula graph is connected, its components can be canonically ordered by bridges (Definition 3.1.5). Maximality of the strongly connected components and trees then guarantees the order defined to be a tree order. □

To simplify the forthcoming algorithm, all tree components (which will not play a significant rôle) are collated with other components. Thus, we define a coarser partition, based on Lemma 3.2.11. A graphical example is shown in Figure 3.4.

Theorem 3.2.12
Let $\varphi \in L^1_\mu$ be an alternation-free μ-calculus formula and $\mathcal{G}(\varphi) = (Q, \to, \lambda)$ be its graph representation. Then there exists a set of components $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ such that the following holds:

1. The set $\mathcal{Q}$ is a pairwise disjoint partition of Q, that is $Q = \bigcup \mathcal{Q}$ and for all $i, j \in \{1, \ldots, m\}$, different indexes $i \neq j$ imply disjoint sets $Q_i \cap Q_j = \varnothing$.

2. Every subgraph induced by $Q_i \in \mathcal{Q}$ either

   i. contains only μ-cycles (we call the corresponding subgraph a *μ-component*), or

   ii. contains only ν-cycles (we call the corresponding subgraph a *ν-component*).

3. There exists a tree order $\leq$ on $\mathcal{Q}$ such that for every edge $q \to q'$ with $q \in Q_i$ and $q' \in Q_j$, we have $Q_i \leq Q_j$. More specifically, edges from a node in Q_i lead to

configurations in either the same Q_i or a child Q_j. Without loss of generality, we may assume that $Q_i \leq Q_j$ implies $i \leq j$, so that smaller components with regard to $\leq$ have lower indexes.

For the graphical representation in Figure 3.4, our components are enclosed by a (red) dashed line and the latter by a (green) dotted line. Note that in contrast to Lemma 3.2.11 components are no longer necessarily *strongly* connected. For formulas without any fixpoint formula, we will get a single component which we may arbitrarily name μ-component, for simplification of matters.

Proof. We consider again the formula graph $\mathcal{G}(\varphi) = (Q, \rightarrow, \lambda)$ of φ. With Theorem 3.2.12, there exist components $Q'_1, \ldots, Q'_n$ which can be partially ordered by $\leq$. They are either maximal strongly connected components or trees. Furthermore, let $\lessdot$ denote the cover relation of $\leq$.

The idea is that strongly connected components can be collated with their child components that are trees. Formally, let $i_1, \ldots, i_m \in \{1, \ldots, m\}$ such that $Q'_{i_1}, \ldots, Q'_{i_m}$ comprise exactly those components Q'_i that are maximal strongly connected components. For $j \in \{1, \ldots, m\}$ we define collated components Q_j as:

$$Q_j := Q'_{i_j} \cup \bigcup \{Q'_k \mid Q'_{i_j} \lessdot Q'_k \text{ and } Q'_k \text{ is a tree}\}$$

Furthermore, if the smallest (with regard to $\leq$) component Q'_1 is a tree, then it has not yet been selected for inclusion in some Q_j, thus we arbitrarily add it to the top-most collated component:

$$Q_1 := Q_1 \cup Q'_1$$

As we collated strongly connected components only with tree components, but never with other strongly connected components, Cases (2i) and (2ii) trivially hold due to alternation-freeness of φ. □

Note that for the presented partition it is essential that Lemma 3.2.11 provides a tree order on the components. In Section 3.4.4 we will discuss an alternative definition of a formula graph which yields partially ordered components and, as we will see, a slightly different parallel algorithm.

Remark 3.2.13
The time complexity of computing the decomposition of the formula graph is linear with respect to the formula length $|\varphi|$. Thus, we can establish a labelling for every subformula $\psi \in \mathrm{Sub}(\varphi)$ with its *component number* i if $\psi \in Q_i$ within linear time. The component index can be made unique by enlarging the partial order of the components to a total order in some deterministic way.

3.2.3. Complexity of Model Checking for L^1_μ

Before starting to think about a concrete algorithm, we should consider theoretical limitations of its applicability, i.e. its complexity.

In this section, we recall the notion of model checking and show that this problem for the studied fragment is P-hard, strengthening a similar result by Zhang et al. [99] where the propositional μ-calculus is employed.

In complexity theory, it is a well-accepted view that problems within *Nick's Class* ($\mathcal{NC}$) admit promising parallel computing algorithms [81]. $\mathcal{NC}$ is based on the Boolean Circuit model for computation and describes the problems computable in polylogarithmic time with polynomially many processors. It can be shown that $\mathcal{NC}$ is contained in P. Problems outside of $\mathcal{NC}$ are consequently considered to be inherently sequential. However, it is not known whether $\mathcal{NC}$=P. If not, then especially P-complete problems cannot be in $\mathcal{NC}$. Hence, P-complete problems are called *inherently sequential* [81]. In other words, we show that model checking for L^1_μ-formulas is inherently sequential.

Given a transition system $\mathcal{T}$ and a formula φ, the *model checking problem* is the question whether $\mathcal{T}$ satisfies φ, that is whether $\mathcal{T} \models \varphi$. The *combined complexity* of the model checking problem is its complexity with respect to the product of the size of the transition system and the size of the formula. The *program complexity* of the μ-calculus is the complexity of the model checking problem only with respect to the size of the transition system $\mathcal{T}$.

Zhang et al. [99] showed that the combined complexity for the alternation free μ-calculus is P-complete and, for a version of the alternation free μ-calculus employing two actions, that its program complexity is P-complete. Kupfermann et al. [61] show the latter result by using a formula with two propositions. We strengthen both results by employing neither propositions nor any action labelling.

Lemma 3.2.14
The program complexity of the alternation free μ-calculus is P-hard.

Proof (Leucker [14]). We reduce the P-complete *Game Problem* [47] to checking a formula of the alternation free μ-calculus with respect to a corresponding labelled transition system. A two-player game is a tuple $G = (P_1, P_2, M, W_0, s)$. Here, P_1 and P_2, with $P_1 \cap P_2 = \varnothing$, are positions, in which it is the turn of Player 1 or Player 2, respectively. $M \subseteq (P_1 \times P_2) \cup (P_2 \times P_1)$ is the set of moves the respective player can make. $W_0 \subseteq P_1 \cup P_2$ denotes the *succeeding positions* and $s \in P_1$ the *starting position*. The players move alternately beginning with Player 1. We call $x \in P_1 \cup P_2$ *winning* iff either x is succeeding ($x \in W_0$), or, $x \in P_1$ and there is a winning $y \in P_2$ such that $(x, y) \in M$, or $x \in P_2$ and for all $(x, y) \in M$, y is winning. The Game Problem is the question whether s is winning.

Corresponding to this, we define a transition system $\mathcal{T}_G = (P_1 \cup P_2, T)$ by $T = (M - \{(p, q) \in M \mid p \in W_0\}) \cup \{(p, p) \in P_1 \times P_1 \mid p \notin W_0$ and there is no transition

from p in $M\}$.

T is defined such that every deadlock state (state with no outgoing edges), is either a state of W_0, or a state of P_2 in which Player 2 is not able to move. Hence, deadlocks are winning. They can be characterized in the μ-calculus by $\varphi_{W_0} = [-]\texttt{false}$ where a formula $[-]\varphi$ indicates that φ is satisfied in all successor states. Further winning positions for Player 1 are states of P_1 such that there is a successor state (in P_2) whose direct successors (in P_1) are all winning. Hence, the formula $\varphi = \mu X.(\langle - \rangle [-] X \vee \varphi_{W_0})$ is satisfied in exactly those positions of P_1 which are winning where $\langle - \rangle \varphi$ guarantees the existence of a successor state in which φ holds. Note that φ may be satisfied in further positions of P_2 which does not bother us. We conclude that s is winning in the game $G = (P_1, P_2, M, W_0, s)$ iff $\mathcal{T}_G, s \models \varphi$.

The construction of the transition system can be done within logarithmic amount of space, viz LOGSPACE. Note that we do not make use of propositions. Furthermore, we manage without actions at all by slightly adapting the modal fragment of our logic. □

Together with a linear-time algorithm from Section 3.3.1 or from Cleaveland and Steffen's work [29], we can state:

Theorem 3.2.15 ([14])
Model checking for the alternation-free μ-calculus is inherently sequential with respect to the combined complexity as well as the program complexity.

As consequence of the previous theorem, our enthusiasm is limited when it comes to finding a (theoretically) good parallel model checking algorithm. All we can possibly hope for is to find a linear-time algorithm (unless $\mathcal{NC}$ equals P).

During the next sections, we will present such a parallel model-checking algorithm and several of its variations which have linear or quadratic time complexity. As we will see in Section 4.1, an implementation of our algorithms is applicable to many practical problems.

3.2.4. Model-checking Games for the μ-calculus

In this section, we recall a characterization of the model checking problem in terms of games. Games are an easy-to-understand formalism to deal with model checking the μ-calculus. A reduction to *parity games* was first noted by Emerson et al. [41]. We use, however, Stirling's presentation [91], as it is more pictorial and thus easier to understand.

As we will see, determining the satisfaction of a formula is reduced to coloring a structure called *game graph*. We will explain that the decomposition of a formula's graph induces a decomposition of the game graph. The latter will simplify our sequential as well a parallel coloring algorithm. Notice that our definition deviates slightly from Stirling's original approach. We do so to obtain a tree-like decomposition of the game graph instead of a DAG-like decomposition. We refer to Section 3.4.4 for a discussion of the advantages and disadvantages of this approach.

Let in the following $\mathcal{T} = (S, \rightarrow, \Sigma, s_0, P)$ be a transition system and let $\varphi \in L_\mu$ be a formula over Var, $Prop$, and Σ with formula graph $\mathcal{G}(\varphi) = (Q, \rightarrow_\varphi, \lambda)$. We can then define the *model checking game* $\Gamma(\mathcal{T}, \varphi)$ for a given transition system $\mathcal{T}$ and a formula φ.

Definition 3.2.16 (Game Configuration)
A *configuration* $C = (s', \pi)$ of a model checking game $\Gamma(\mathcal{T}, \varphi)$ is a pair consisting of a state $s' \in S$ from the transition system and a position $\pi \in Q$ in the formula graph $\mathcal{G}(\varphi) = (Q, \rightarrow_\varphi, \lambda)$.

Definition 3.2.17 (Game Board)
The *game board* is defined as the set of all possible configurations $S \times Q$.

The game $\Gamma(\mathcal{T}, \varphi)$ is played by two players, namely $\forall$*belard* (the pessimist), who wants to show that $\mathcal{T} \models \varphi$ does *not* hold, whereas $\exists$*loise* (the optimist) wants to show the opposite.

Definition 3.2.18 (Move)
A game *move* from some configuration C to configuration C' is denoted as $C \Rightarrow_P C'$, where P specifies which player has determined this move. $\forall$belard makes universal $\Rightarrow_\forall$-*moves*, $\exists$loise makes existential $\Rightarrow_\exists$-*moves*. We write $\Rightarrow$ instead of $\Rightarrow_P$ if we abstract from the player.

Unlike traditional Ehrenfeucht-Fraïssé games, players do not have to move alternately. The next turn is not determined by the player but by the second part of a configuration, that is in effect a subformula of φ.

Definition 3.2.19 (Play)
A single *play* $G^{\mathcal{T}}(s, \varphi)$ of a game $\Gamma(\mathcal{T}, \varphi)$ is a (possibly infinite) sequence $C_0 \Rightarrow_{P_0}$

$C_1 \Rightarrow_{P_1} C_2 \Rightarrow_{P_2} \ldots$ of configurations, with initial configuration $C_0 = (s, \epsilon)$, and for all configurations $C_i \in S \times Q$. We will elide the transition system $\mathcal{T}$ and write $G(s, \varphi)$ instead if the context is unambiguous.

The model checking game $\Gamma(\mathcal{T}, \varphi)$ is then given by all its plays, that is, all possible sequences $C_0 \Rightarrow_{P_0} \ldots$ which are played in accordance to the rules presented in the forthcoming definition.

Definition 3.2.20 (Rules)
For the ith configuration $C_i = (s, \pi)$, we state the game rules, telling when player P_i can move to a successor configuration C_{i+1}, thus $C_i \Rightarrow_{P_i} C_{i+1}$:

1. If $\lambda(\pi) = \texttt{false}$, then the play ends.
2. If $\lambda(\pi) = \psi_1 \wedge \psi_2$, then ∀belard chooses $j = 1$ or $j = 2$, and $C_{i+1} = (s, \pi j)$.
3. If $\lambda(\pi) = [K]\psi$, then ∀belard chooses a transition $s \xrightarrow{a} t$ with an action $a \in K$ and $C_{i+1} = (t, \pi 1)$.
4. If $\lambda(\pi) = \nu X.\, \Psi_X$, then $C_{i+1} = (s, \pi 1)$.
5. If $\lambda(\pi) = \texttt{true}$, then the play ends.
6. If $\lambda(\pi) = \psi_1 \vee \psi_2$, then ∃loise chooses $j = 1$ or $j = 2$, and $C_{i+1} = (s, \pi j)$.
7. If $\lambda(\pi) = \langle K \rangle \psi$, then ∃loise chooses a transition $s \xrightarrow{a} t$ with an action $a \in K$ and $C_{i+1} = (t, \pi 1)$.
8. If $\lambda(\pi) = \mu X.\, \Psi_X$, then $C_{i+1} = (s, \pi 1)$.
9. If $\lambda(\pi) = X$, then there exists a single edge $\pi \rightarrow_\varphi \pi'$ in the formula graph, such that π' denotes the position of the fixpoint formula $\lambda(\pi') = \sigma X.\, \Psi_X$ identified by X. Hence $C_{i+1} = (s, \pi')$.
10. If $\lambda(\pi) = p$, then the play ends.
11. If $\lambda(\pi) = \neg p$, then the play ends.

As moves 1,4,5, and 8–11 are deterministic, no player needs to be charged with them. With regard to the forthcoming winning strategies and our algorithm, we will speak of ∀*belard-moves* in cases 1–4, 9 and 10 if $\sigma = \mu$, and ∃*loise-moves* in all other cases. A configuration C_i is then called ∀-configuration or ∃-configuration, respectively.

So far, we only laid out the rules according to which a game must be played. What remains is to specify which player wins a play.

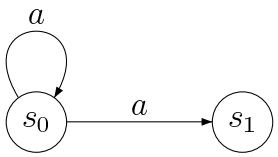


Figure 3.5.: A transition system to demonstrate different plays in a model checking game: on all paths, only a-actions can be observed.

Before we state winning conditions formally, it is instructive to consider which kinds of situations can arise. For example, we present three possible plays for the game given by the transition system in Figure 3.5 and the formula

$$\varphi = \mu X.\, \langle - \rangle X \vee \langle a \rangle \mathtt{true}$$

which expresses the possibility to perform an a-action eventually. For presentational reasons, we will substitute a position π in the formula graph with the actual subformula $\lambda(\pi)$ it represents.

- $G_1(s_0, \varphi) = (s_0, \varphi) \Rightarrow_\exists (s_0, \langle - \rangle X \vee \langle a \rangle \mathtt{true})$
  $\Rightarrow_\exists (s_0, \langle a \rangle \mathtt{true})$
  $\Rightarrow_\exists (s_1, \mathtt{true})$

- $G_2(s_0, \varphi) = (s_0, \varphi) \Rightarrow_\exists (s_0, \langle - \rangle X \vee \langle a \rangle \mathtt{true})$
  $\Rightarrow_\exists (s_0, \langle - \rangle X)$
  $\Rightarrow_\exists (s_1, X)$
  $\Rightarrow_\forall (s_1, \varphi)$
  $\Rightarrow_\forall (s_1, \langle - \rangle X \vee \langle a \rangle \mathtt{true})$
  $\Rightarrow_\exists (s_1, \langle a \rangle \mathtt{true})$

- $G_3(s_0, \varphi) = (s_0, \varphi) \Rightarrow_\exists (s_0, \langle - \rangle X \vee \langle a \rangle \mathtt{true})$
  $\Rightarrow_\exists (s_0, \langle - \rangle X)$
  $\Rightarrow_\exists (s_0, X)$
  $\Rightarrow_\forall (s_0, \varphi)$
  $\Rightarrow_\exists \ldots$

The first play G_1 clearly is won by $\exists$loise because she forced it to end in a `true`-configuration. It is also clear that $\forall$belard wins G_2, because $\exists$loise became stuck as there is no a-action possible in state s_1.

We must now ask the question: Who wins the game G_3 that has an infinite loop? As the game contains infinitely many X-configurations it is equivalent to an infinite

unfolding of a minimal fixpoint formula. That means the fixpoint could not be computed (otherwise the game should be of finite length). Minimal fixpoints are special (second-order) $\exists$-quantifiers. So $\exists$loise—who should do correct existential moves—failed. Therefore $\forall$belard wins G_3.

A configuration is called *terminal* if no (further) move is possible. A play $C_0 \Rightarrow \cdots$ is called *maximal* iff it is infinite or it ends in a terminal configuration. The *winner* of a maximal play is then defined in the following way.

Definition 3.2.21 (Winning Condition)
We distinguish between finite and infinite maximal plays. If the play is finite, thus ending in a configuration (s, π), then $\forall$belard wins a play G, iff

- $\lambda(\pi) = \texttt{false}$,
- $\lambda(\pi) = \langle K \rangle \psi$,[2] or
- $\lambda(\pi) = p$ and $s \notin P(p)$
- $\lambda(\pi) = \neg p$ and $s \in P(p)$

Dually, $\exists$loise wins a play G, iff

- $\lambda(\pi) = \texttt{true}$,
- $\lambda(\pi) = [K] \psi$,[2] or
- $\lambda(\pi) = p$ and $s \in P(p)$
- $\lambda(\pi) = \neg p$ and $s \notin P(p)$

An infinite play is won by $\forall$belard if the outermost fixpoint which is unwinded infinitely often is a μ-fixpoint. Otherwise, the outermost fixpoint which is unwinded infinitely often is a ν-fixpoint and $\exists$loise wins the game.

As we have seen, given a transition system and a formula, there are several possible plays and these not necessarily have the same winner. Thus, we must ensure that our players play as good as they can, according to some *strategy*.

Formally, a *strategy* is a set of rules for a player P telling her or him how to move in the current configuration. It is called *history-free*, if the strategy only depends on the current configuration without considering previous moves. A *winning strategy* now guarantees that the play which P plays according to the rules will be won by P.

Stirling showed that the model checking problem for the μ-calculus is equivalent to finding a history-free winning strategy for one of the players [91]: Let $\mathcal{T}$ be a transition

[2] Note that due to maximality of G we have $\nexists s' : s \xrightarrow{a} s'$ for any $a \in K$.

system with starting state s, and let φ be a μ-calculus formula. $\mathcal{T} \models \varphi$ implies that $\exists$loise has a history-free winning strategy starting at (s_0, φ), and $\mathcal{T} \not\models \varphi$ implies that $\forall$belard has a history-free winning strategy for all plays starting at (s_0, φ). As a formula either holds or is falsified, this result also implies that model checking games are *determined*, that is for every game *either* $\forall$belard *or* $\exists$loise has a winning strategy.

We will devote the coming sections to formalize how to compute winning strategies in various ways.

Remark 3.2.22
In the coming chapters we will usually not deal with a game $\Gamma(\mathcal{T}, \varphi)$ directly. Instead we focus on its *game graph* $\mathcal{G} = (Q, E)$, with Q being the set of all configurations C_i in all plays of $\Gamma(\mathcal{T},)$, and E being the edge relation induced by all possible rule-conforming moves $\Rightarrow_{P_i}$.

As the naming already suggests, there is strong similarity of a game graph to the corresponding formula graph $\mathcal{G}(\varphi)$. We will exploit this similarity for our algorithms.

The rules from Definition 3.2.20 admit observations about the structure of a game graph.

Remark 3.2.23
In a move $(s, \pi) \Rightarrow_P (s', \pi')$ changes of the state part s to s' are driven by the formula part π, that is only through application of rules 3.2.20(3) and (7) it might happen at all that $s \neq s'$. Furthermore, for all possible moves it holds that $\pi \neq \pi'$, meaning there is always progress in the second part of a configuration, and thus the formula graph does not contain any trivial cycles. In consequence, the game graph does not contain trivial cycles either.

3.3. Winning L_μ^1-games

In the following, we concentrate on the alternation-free fragment L_μ^1 of the μ-calculus. This fragment admits a characterization of a game graph $\mathcal{G}$ for a transition system $\mathcal{T}$ and a formula φ in terms of the formula graph $\mathcal{G}(\varphi)$ which will lead us to intuitive sequential and parallel algorithms to compute the winning strategy of a player, thus solving the model checking problem whether $\mathcal{T} \models \varphi$.

In order to get there, we first exploit the resemblance of the game graph $\mathcal{G}$ and its underlying formula graph $\mathcal{G}(\varphi)$ by lifting the notions we have established previously from one structure to the other.

Theorem 3.3.1
Let $\mathcal{T}$ be a labelled transition system and let $\varphi \in L_\mu^1$ be a formula of the alternation-free μ-calculus. Furthermore, let $\mathcal{G} = (Q, E)$ be their resulting game graph. Then there exists a set of components $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ for Q such that properties from components of the formula graph $\mathcal{G}(\varphi)$ carry over, namely:

1. The set $\mathcal{Q}$ is a partition of Q, that is $Q = \bigcup \mathcal{Q}$ and $\forall Q_i, Q_j \in \{1, \ldots, m\} : \; i \neq j \Rightarrow Q_i \cap Q_j = \varnothing$.

2. In every subgraph induced by $Q_i \in \mathcal{Q}$ either μ-formulas and no ν-formulas are unwinded or ν-formulas and no μ-formulas. Just like components in the underlying formula graph, we call a component $Q_i \in \mathcal{Q}$ respectively μ-component or ν-component.

3. There exists a tree order $\leq$ on $\mathcal{Q}$ such that for every game-graph edge $(C, C') \in E$ with $C \in Q_i$ and $C' \in Q_j$, we have $Q_i \leq Q_j$. More specifically, edges from a node in Q_i lead to configurations in either the same Q_i or a child Q_j. Without loss of generality, we may assume that $Q_i \leq Q_j$ implies $i \leq j$, so that smaller components with regard to $\leq$ have lower indexes.

Proof. By Theorem 3.2.12, the formula graph of φ admits a decomposition into either μ- or ν-components $\mathcal{Q}' = \{Q'_1, \ldots, Q'_m\}$. We can then show the desired properties:

1. Let each component Q_i be the set of configurations $\{(s, \pi) \mid \pi \in Q'_i\}$ with formulas of component number i in the formula graph's partition $\mathcal{Q}'$. Thus, we lifted the formula graph's partition onto the game graph, resulting in partition $\mathcal{Q}$.

2. As the classification of a component as σ-component is solely based on the formula part of a configuration, it can be carried over directly from the formula graph.

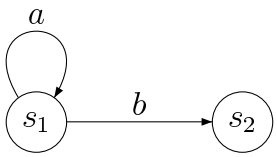


Figure 3.6.: A transition system

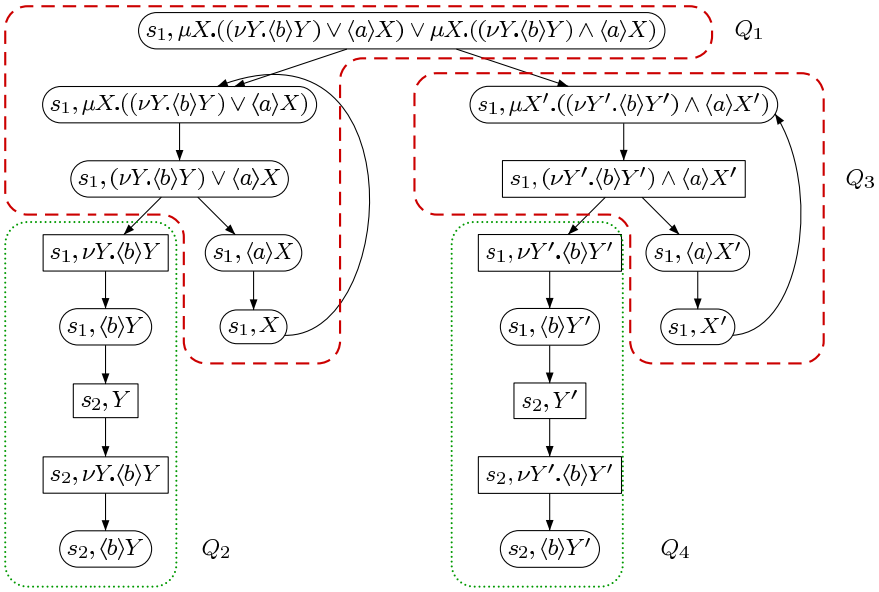


Figure 3.7.: A game graph and its partition.

3. The tree order $\leq$ on $\mathcal{Q}$, which is due to alternation-freeness, can be lifted from the formula-graph partition $\mathcal{Q}'$, such that for all $i, j \in \{1, \ldots, m\}$, $Q'_i \leq_{\mathcal{Q}'} Q'_j$ implies $Q_i \leq Q_j$.

□

Figure 3.7 shows the game graph for the transition system in Figure 3.6 and a formula $\varphi = \mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X') \in L^1_\mu$ which we already used in Figure 3.4, along with its decomposition into components according to the formula graph of φ.

Note that ∀belard-configurations are marked by rectangular boxes while ∃loise-configurations are drawn as ovals. The dashed and dotted lines identify μ-components and respectively ν-components. In all forthcoming examples, we write the formula instead of its position in the second component of a configuration, to simplify our presentation.

Let us fix the decomposition of the game graph shown in the previous proof in the following definition:

Definition 3.3.2 (Canonical Decomposition, Escape and Initial Configuration)
Let $\mathcal{T}$ be a labelled transition system and let φ be a formula of the alternation-free μ-calculus. Furthermore, let $\mathcal{G} = (Q, E)$ be their game graph.

- The *canonical decomposition* of $\mathcal{G}$ is the decomposition according to Theorem 3.3.1 into $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ which are tree-ordered by $\leq$.
- Let $Q_{j_1}, \ldots, Q_{j_{k_i}}$ be the components that are children of Q_i. The *escape configurations* of a component Q_i (denoted by $\lfloor Q_i \rfloor$) are the configurations which are in a child component and are successor configurations of a configuration in Q_i. That is:

$$\lfloor Q_i \rfloor = \{C \in Q_j \mid Q_j \text{ is a child of } Q_i \text{ and } \exists C' \in Q_i \text{ such that } (C', C) \in E\}$$

- The *initial configurations* $\lceil Q_i \rceil$ of a component Q_i are those configurations which are escape configurations for a parent component, and for Q_1 additionally the initial configuration C_0 of the game graph.
- In compliance with the terminology of formula graphs, we define the *component number* of a configuration C of the game graph as the (unique) index i of a component Q_i that contains the configuration: $C \in Q_i$.

Remark 3.3.3
The formula part of an escape configuration is labelled by a fixpoint formula. Furthermore, a component has no escape configurations, that is $\lfloor Q_i \rfloor = \varnothing$, iff it is a leaf with respect to the tree order $\leq$.

As components are tree-ordered by $\leq$, we can guarantee that every infinite play eventually gets trapped within some component Q_i, that is for a play $C_0 \Rightarrow \ldots \Rightarrow C_k \Rightarrow \ldots$ there exists a configuration C_k such that later in the play all configurations $C_j, j > k$ belong to component Q_i.

As example, we pick the case that this *trapping component* Q_i is a ν-component. Hence, it contains a configuration with a ν-fixpoint which appears infinitely often in the play, and according to the winning conditions in Definition 3.2.21, $\exists$loise wins this play. Thus, $\exists$loise is interested keeping the play inside a ν-component. Consequently, to avoid losing the play, $\forall$belard would be well-advised to make it his strategy to leave ν-components if at all possible. Thus, he must try moving to an escape configuration. With a dual argument, $\exists$loise must try to leave μ-components via their escape configurations.

Remark 3.3.4
The number of components of a game graph's canonical decomposition is identical to

the number of the components of the graph of the formula according to Theorem 3.2.12. Also, the component number of a configuration is identical to the number of the component of its formula label (which is defined in the obvious manner). Thus, once we computed the component number of a (sub)formula as described in Remark 3.2.13, it is a constant operation to check the component number of a configuration.

3.3.1. Sequential Coloring Algorithms

In this subsection, we present two sequential approaches for determining winning strategies, thereby solving the model checking problem. Note that we develop these algorithms foremostly as an intermediate step towards efficiently parallelizable algorithms, which we will detail in the next section. For an actual sequential implementation, though, a different algorithm would be preferable, for example the one presented by Lange [62], as it performs slightly faster in practice and is easier to implement.

The basic idea of both algorithms is labelling a configuration C with colors $green$ or red, depending on whether ∃loise or ∀belard has a winning strategy for the game starting in this configuration C. Furthermore, they both employ the canonical decomposition of the game graph according to Definition 3.3.2. They differ however in the order the components are processed. The first algorithm proceeds bottom-up, the second top-down.

Figure 3.7 shows a game graph and its canonical decomposition into disjoint sets Q_i that are μ- or ν-components as per Definition 3.3.2. Furthermore, for every $C \in Q_i$ and $C' \in Q_j$ for which there is a possible move from C to C', we have that $i \leq j$. Thus, every infinite play gets trapped within a single Q_i, and the winner depends on the fact whether a μ-formula or a ν-formula is unwinded in Q_i. For the example shown in Figure 3.7, we obtain four components, $Q_1, \ldots, Q_4$.

Coloring Bottom-up

First, we discuss how to color a single component. Let Q_i be a component of the canonical decomposition. To simplify the presentation, assume that Q_i is a μ-component. The forthcoming explanation can be dualized for ν-components.

Let $\lfloor Q_i \rfloor$ denote the set of escape configurations and assume that every escape configuration $C \in \lfloor Q_i \rfloor$ is either labelled $green$ or red, expressing that either ∃loise or ∀belard has a winning strategy from this configuration, respectively. All other configurations have no color initially, to which we may refer to as $white$.

Every play starting in some configuration $C \in Q_i$ will either

1. eventually reach an escape configuration and never touch a configuration of Q_i again,
2. will end in a terminal configuration within Q_i, or
3. will go on infinitely within Q_i.

In the first case, the winner is determined by the color of the escape configuration. In the second case, the terminal configuration signals whether ∃loise or ∀belard has won. The last case makes ∀belard a winner, as a μ-formula is unwinded infinitely often.

Algorithm 3.1 initializeConfiguration($conf$)

Initialize counts for configuration $conf$.

```
1: if conf is ∀belard-configuration then
2:     count(conf, RED) := 1
3:     count(conf, GREEN) := |post(conf)|
4: else    /* conf is ∃loise-configuration */
5:     count(conf, RED) := |post(conf)|
6:     count(conf, GREEN) := 1
7: end if
```

Algorithm 3.2 color($conf$)

Compute color for configuration $conf$.

```
1: if ∃color ∈ {RED, GREEN} : count(conf, color) = 0 then
2:     return color
3: else
4:     return λ(conf)
5: end if
```

The second case justifies coloring every terminal configuration within Q_i according to the winning conditions from Definition 3.2.21 in the following way. If the formula part of the configuration is `true`, a universal modality formula $[K]\varphi$, a proposition p that is valid in the current state, or a negated proposition $\neg p$ not valid in the current state, then the configuration is colored with $green$. Otherwise, the formula part is `false`, an existential modality $\langle K\rangle\varphi$, a non-satisfied proposition, or a satisfied negated proposition and the configuration is colored with red.

Once a configuration $C_k \in Q_i \cup \lfloor Q_i \rfloor$ is labelled red or $green$, its predecessors are labelled if possible. Thus, an ∀belard-configuration C is labelled red if C_k is red, and labelled $green$ if all successors $\{C_j \mid C \Rightarrow C_j\}$, are $green$. In Algorithm 3.1, we use two counters for this book-keeping—one for each color—to determine the number of successors of C_k which need to be colored before C_k itself can be colored. A color can be determined, if one of the counters reaches zero (Algorithm 3.2).

If C is an ∃loise-configuration, it is treated dually, that is the rôles of the colors are swapped. If a predecessor configuration C has obtained a new color through this process, the labelling is propagated further in the same way. The backwards color propagation within a single component is detailed in Algorithm 3.3.

The coloring process allows the following insight:

Lemma 3.3.5 (Color Stability)
Once a configuration obtained a color, it remains *stable*, that is, the color never changes.

Algorithm 3.3 colorizeComponent(Q_j), sequential version

Colorize configurations of component Q_j.

Require: $Work := \{\textsc{Color}(pred, \lambda(conf)) \mid pred \in pre(conf), conf \in Q_j \cup \lfloor Q_j \rfloor, \lambda(conf) \neq \textsc{White}\}$

```
1:  while Work ≠ ∅ do
2:    Color(conf, color) := get(Work)
3:    decrement count(conf, color)    /* update color information */
4:    color' := color(conf)
5:    if color' ≠ λ(conf) then
6:      λ(conf) := color'
7:      for each pred ∈ pre(conf) ∩ Q_j do
8:            /* only work on current component */
9:        put Color(pred, λ(conf)), Work
10:     end for
11:   end if
12: end while
```

As color labels are stable, and a component contain only finitely many configurations, the following is immediately obvious:

Corollary 3.3.6
The coloring process is terminating.

However, this labelling process can leave some configurations of the component Q_i uncolored. Let us now understand that all remaining uncolored configurations of a μ-component can be labelled red.

Theorem 3.3.7
For any game starting in a configuration $C_k \in Q_i$ without a color, where Q_i is a μ-component, ∀belard has a winning strategy for a game starting in C_k. Dually, ∃loise has a winning strategy for any game starting in a configuration $C_k \in Q_i$ without a color, where Q_i is a μ-component.

Proof. We will direct our focus to ∀belard's case only, as it can be dualized for ∃loise easily. Firstly, check that every uncolored configuration has at least one uncolored successor configuration. ∀belard's strategy will be choosing any one uncolored successor in this situation. Then he will win every play. Every uncolored ∃loise-configuration has red or uncolored successors, so ∃loise has the choice to move to configurations which are winning for ∀belard or to move to an uncolored configuration. ∀belard will choose in an uncolored configuration an uncolored successor, or, if ∃loise has moved to a red configuration then he will choose a red successor.

Summing up, every play will either end in a red terminal configuration, lead to a red escape configuration from which ∀belard has a winning strategy, or will go on infinitely often within Q_i, and thus ∀belard wins in either case. □

The previous theorem is the crucial observation allowing a powerful parallel version of this algorithm. Observe that we do not need to employ any cycle detection algorithm[3] in the labelling process. We know that the described backward color propagation process is leaving only those configurations uncolored that are on or lead to a cycle which furthermore can be controlled by ∀belard.

Our first sequential algorithm now processes the components in a bottom-up fashion. First, leaf components which have no escape configurations are considered and colored. Now, for any parent component the escape configurations are labelled and, again, our procedure will color the component.

Let us turn back to our example shown in Figure 3.7. It is partitioned into four components, $Q_1, \ldots Q_4$. One leaf component is Q_2. The single terminal configuration $(s_2, \langle b \rangle Y)$ requires ∃loise to present a b-successor of s_2. However, in the underlying transition system, there is no successor. Thus, the configuration will be labelled red. Now, propagating the colors to the predecessor configurations will color every configuration of Q_2 with red.

The other leaf component Q_4 in our graph will be treated in the same manner as Q_2.

The next component to handle with respect to our tree order is Q_3. It has the single escape configuration $(s_1, \nu Y'.\langle b \rangle Y')$, which is already colored red. This color is propagated to $(s_1, \nu Y'.\langle b \rangle Y' \wedge \langle a \rangle X')$, which now is colored red. Further propagation will color the whole component Q_3 with red.

Next, we proceed with Q_1. $(s_1, \nu Y.\langle b \rangle Y)$ propagates red to $(s_1, \nu Y.\langle b \rangle Y \vee \langle a \rangle X)$. Since the latter is an ∃loise-configuration, it remains uncolored. A similar situation occurs for the propagation due to $(s_1, \mu X'.((\nu Y'.\langle b \rangle Y') \wedge \langle a \rangle X'))$. Thus, all color information is propagated within Q_1. The current situation is depicted in Figure 3.8 in which red configurations are filled with ■. Now, the second phase of coloring a component comes into play. All remaining configurations will be labelled red since Q_1 is a μ-component. Thus, ∀belard has a winning strategy for the presented game and we know that the underlying formula is not valid in the initial state of the transition system.

Complexity It is a simple matter to see that the previous labelling algorithm has a linear running time (in the worst case) with respect to the size of the game graph. The size of the game graph is bounded by the product of size of the underlying transition system $|\mathcal{T}|$ and the length of the formula $|\varphi|$, that is, it is bounded by $|\mathcal{T}| \times |\varphi|$. However, only the part of the transition system *related* to the underlying formula has to be

[3] cycle dectection is costly in a parallel or distributed setting

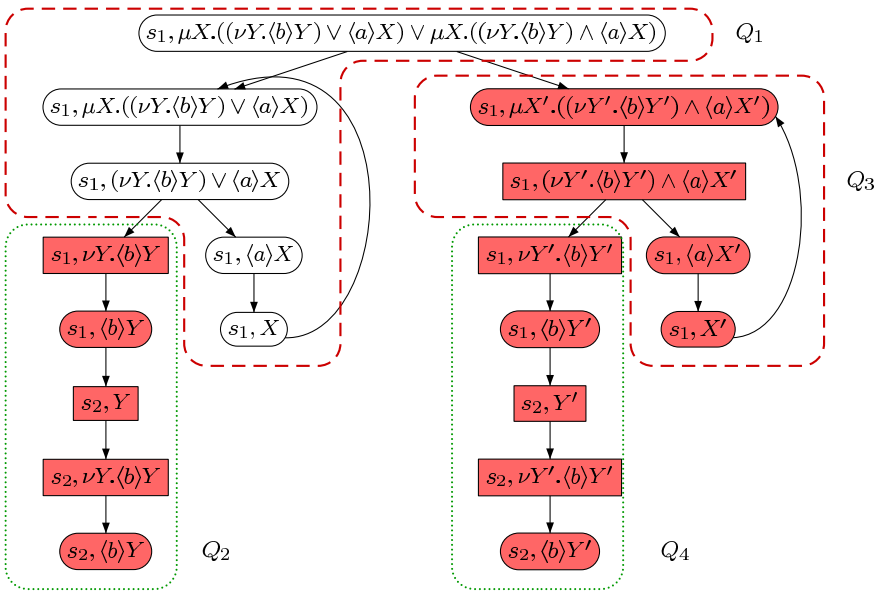


Figure 3.8.: The situation before the second phase

considered. For example, checking $\langle K\rangle\varphi$ in a state s requires only to look for a successor reachable by actions $a \in K$ that satisfies φ. All successors reachable by different actions $a' \notin K$ need not be considered. While in the worst case, the whole transition system has to be considered checking a formula, only a part of the system has to be generated in typical examples. Thus, we can call our algorithm to be *local* or *on-the-fly*.

Coloring Top-down

For the second algorithm, let us assume that the game graph is again partitioned into components Q_i which form a tree. In order to get a result while considering as few components as possible, the algorithm will process the components in a top-down manner.

Again, we first discuss how to color a single component. Let Q_i be a component of the canonical decomposition. We assume that Q_i is a μ-component recalling that the forthcoming explanation can be dualized for ν-components. Let $\lfloor Q_i \rfloor$ denote its escape configurations. However, we will *not* assume that all escape configurations $\lfloor Q_i \rfloor$ are colored already. Still, every play will either

1. eventually reach an escape configuration and never touch a configuration of Q_i again,

2. will end in a terminal configuration within Q_i, or

Algorithm 3.4 $\mathrm{color}(conf)$, top-down version

Compute color for configuration $conf$

```
 1: color' := color(conf)
 2: if color' ≠ WHITE then
 3:    return color'
 4: else if Σ_{c∈{RED,LIGHTRED}} count(conf, c) = 0 then
 5:    return LIGHTRED
 6: else if Σ_{c∈{GREEN,LIGHTGREEN}} count(conf, c) = 0 then
 7:    return LIGHTGREEN
 8: else if conf ∈ ⌊Q_j⌋ then
 9:    if Q_j is a μ-component then
10:       return LIGHTGREEN
11:    else    /* Q_j is a ν-component */
12:       return LIGHTRED
13:    end if
14: else
15:    return λ(conf)
16: end if
```

3. will go on infinitely within Q_i.

Again, the winner of a play is clear in Case (2). Furthermore, if ∃loise has neither a way to reach a winning terminal configuration nor to leave the component, she will loose. So, if she has no chance to reach a winning terminal configuration, the best we can hope for her, is that she indeed has a chance to leave the component successfully. The crucial point of our algorithm is that we initially color all escape configurations of the component under consideration with *lightgreen*, denoting that this configuration is *probably* a winning configuration for ∃loise.

As before, the color information (full as well as light colors) is propagated to predecessor configurations and used for coloring this configuration. That means, an ∀belard-configuration is labelled *red* if one successor is *red*, labelled *lightred*, if no successor is *red* but at least one is *lightred*, labelled *lightgreen*, if all successors are *lightgreen* or *green* and one is indeed *lightgreen*, and labelled *green*, if all successors are *green*. In all other cases, the configuration remains unlabelled. An ∃loise-configuration is treated dually. Note that in cases without light colors involved, these are the same rules as for the bottom-up approach described previously.

In order to keep track of successors colored with any one of the light colors, we introduce two additional counters per configuration, which must be initialized with zero for every configuration in the game graph. To simplify color calculations in Algorithm 3.4, they count negative. Their absolute counter value reflects the number of light-colored

successors of a configuration.

Note that $lightred$ comes only into play for ν-components. If a predecessor obtained a new color, this labelling is propagated further. A case analysis shows the following result.

Lemma 3.3.8 (Color Stability)
Once a configuration obtained a full color, it is *stable*. A light color is only changed to the corresponding full color.

Corollary 3.3.9
The coloring process is terminating.

Again, the labelling process may leave some configurations of Q_i uncolored. Let us now understand, that, just like in the bottom-up coloring process, all remaining uncolored configurations can be labelled red.

Theorem 3.3.10
For any game starting in a configuration without a color, ∀belard has a winning strategy for a game starting in this configuration.

Proof. First, check that every uncolored configuration has at least one uncolored successor configuration. ∀belard's strategy will be any choosing one uncolored successor. Then he will win every play. Every uncolored ∃loise-configuration has red, or uncolored successors, so ∃loise has the choice to move to a configuration which is winning for ∀belard or to move to an uncolored configuration. ∀belard will choose in an uncolored configuration an uncolored successor, or, if ∃loise has moved to a non-terminal red configuration, he will choose a red successor. Summing up, every play will either end in a red terminal configuration, move to a red escape configuration in which ∀belard has a winning strategy, or will stay infinitely often within Q_i and ∀belard wins. □

Again, the previous theorem is crucial and admits a parallel version of this algorithm which will work top-down.

Unlike in the bottom-up case, our component now may contain configurations which are colored with $lightgreen$. However, we cannot guarantee that ∃loise has indeed a winning strategy for games starting in such a configuration. Thus, we remove the color of such a configuration. If the initial configurations of the component are colored, we are done. If not, we have to consider a child component to get further evidence.

Let us turn back to our example shown in Figure 3.7. We introduce the color $white$ to identify uncolored configurations, assuming that initially every configuration has a $white$ color. We start with the root component Q_1. Both escape configurations are initially labelled $lightgreen$ (cf. Figure 3.9[4]).

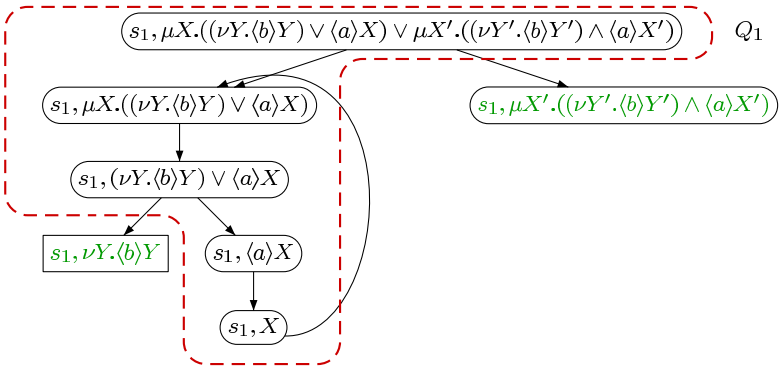


Figure 3.9.: Starting with Q_1.

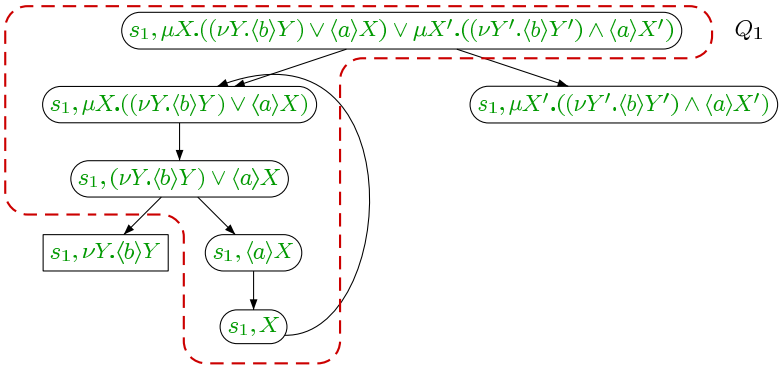


Figure 3.10.: Propagating assumptions.

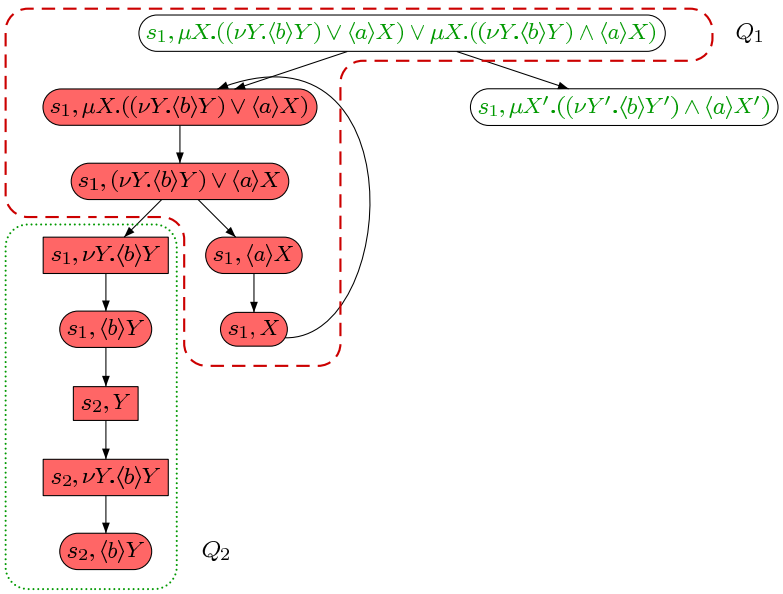


Figure 3.11.: Propagating assumptions for Q_2 and drawing conclusions.

Propagating color information will color every configuration of Q_1 with *lightgreen*, which is shown in Figure 3.10.

The subsequent phase of coloring *white*-configurations with *red* and *lightgreen*-configurations with *white* will turn the whole component to a complete *white* one, so that it looks similar to the one in the beginning. Thus, the assumptions did not help to find a winner. Therefore, we have to check a child component of Q_1. Let us proceed with Q_2. Since there are no escape configurations, the whole component is colored as before. We learn that the *lightgreen* assumption for the initial configuration of Q_2 was too optimistic. Redoing the coloring of Q_1, now with more but still not full information, will color some configurations of Q_1 *red* but will still leave the initial configuration uncolored. Figure 3.12 shows the colored game graph right before recoloring the speculatively *lightgreen* colored configurations back to *white*.

We turn our attention to Q_3. We assume that ∃loise has a chance to leave the component via $(s_1, \nu Y'.\langle b\rangle Y')$. Thus, we color this configuration *lightgreen*. However, the propagation does not influence the preceding ∀belard-configuration. So all remaining configurations are colored *red*. Now, all escape configurations of Q_1 are colored and a further coloring process will color the complete component Q_1 *red*. Note that we saved the time and especially space for considering Q_4. Figure 3.12 shows the colored game graph, again right before recoloring *lightgreen* back to *white*.

[4] *lightgreen* configurations are identified by writing their label in *lightgreen*.

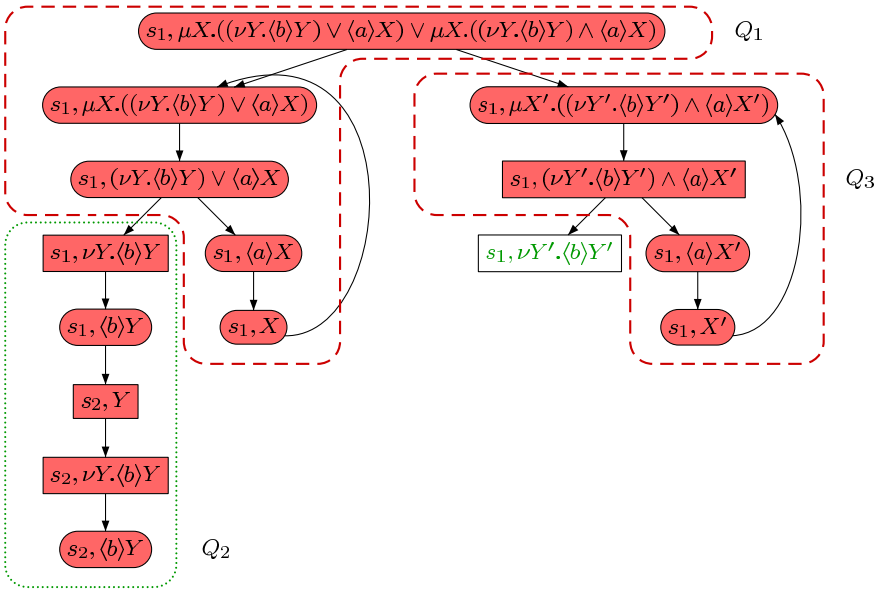


Figure 3.12.: Propagating assumptions for Q_3 and drawing conclusions.

Complexity It is a simple matter to see that the previous labelling algorithm has a run-time bounded by $n \times m$ where n is the number of configurations of the game graph and m is the size of the maximum length of a path from the root component to a leaf component. The latter number is bounded by the nesting of fixpoint formulas which is at most the length of the formula. Thus, we get as an upper bound $s \times l^2$ where s is the size of the underlying transition system and l is the length of the formula. While in the worst case, this complexity is worse than in the bottom-up approach, we found out that the algorithm often detects the truth-value of a formula in a given state much faster. Note, that this algorithm exhibits an even bigger degree of locality than the previous one, that is, even less parts of a game graph need to be considered to determine a solution of the model checking problem.

3.4. Winning Games for L_μ^1-Formulas in Parallel

Given a transition system and an L_μ^1-formula, our approach is both to construct the game graph as well as to determine the color of its configurations in parallel. The main question is how to color a single component in parallel. The tree of components can then be handled either sequentially in a top-down or bottom-up manner, or, if permitted by computing and storage resources, as well in parallel manner. For the moment, however, we concentrate on how to construct and color a single component in parallel.

3.4.1. Distributing the Game Graph

We will first discuss how to construct (a component of) the game graph in parallel. It is obvious that this can be carried out by a typical breadth-first strategy. Given a node q, determine its successors $q_1, \ldots, q_n$. Now, the successors can be processed in the same manner in parallel. However, to obtain a terminating procedure, only exactly the q_i *not* processed before must be expanded. All states generated must be stored within the NOW, and load sharing must be guaranteed. On a shared memory architecture, this does not involve big conceptual problems. For distributed memory machines, however, this is a little bit more difficult.

A first idea might be to distribute the first $q_1, \ldots, q_n$ to the first n processors, and these process the q_i as described before and distribute the successors to the next processors. However, deciding whether a q_i was processed before becomes an expensive operation. Every processor could have processed q_i and should therefore be consulted. In the worst case, for every node, such a broadcast is required. This yields no reasonable algorithm.

A different, often-employed way to store graphs on a distributed memory machine is to divide the graph's adjacency matrix $M \in \{0,1\}^{|Q|\times|Q|}$ into equally sized *blocks* and to store each block on a single processor [87]. This has several advantages. Firstly, the blocks of the matrix can be generated in parallel. Second, given nodes $p, q \in V$, it is easy to check whether there is an edge from p to q, i.e. whether $M_{p,q} = 1$. Since there is a unique location for the block of the matrix containing the value for the pair (p, q), a single communication is needed. Third, every processor gets the same amount of data. This promises a similar load for every processor. Furthermore, we want to mention that several typical graph algorithms can be formulated in terms of matrix operations and admit a good parallelization [87].

However, for our problem this approach cannot be applied. Our resulting adjacency matrix would be sparse and storing it completely would yield too much overhead. Storing only the non-zero entries, on the other hand, does not guarantee an equal load for the processors. But more importantly, the number of nodes of our graph is not known a priori but computed while constructing the graph.[5] Hence, the partition of the whole

[5] In the context of model checking, the transition system is often not given explicitly but expanded at

game graph into blocks cannot be determined in advance.

We follow Ciardo et al. [23], Stern and Dill[90]. First, we employ adjacency lists instead of matrices. Note that we need also links to the predecessors as well as to the successors of a node for the labelling algorithm. Let f be a function mapping the states of the game graph to a processor of our network. Usually, a function in the spirit of a hash-function is used to assign to every state an integer and subsequently its value modulo the number of processors. Then, f determines the location of every state within the network uniquely and without global knowledge. In a breadth-first manner, starting with the initial state q_0 of the game graph, the state space can be constructed in parallel with the help of f in the following way. Given a state q (and possibly some of its direct predecessors), send it to its processor $f(q)$. If q is already in the local store of $f(q)$, then q is reached a second time, hence the procedure stops. If predecessors of q were sent together with q, the list of predecessors is augmented accordingly. If q is not in the local memory of $f(q)$, it is stored there together with the given predecessors as well as all its new computed successors $q_1, \ldots, q_k$. These are sent in the same manner to their (with respect to f) processors, together with the information that q is a direct predecessor. The corresponding processes update their local memory similarly.

In terms of the matrix implementation, for every q_i a unique processor $f(q_i)$ is determined on which the non-zero entries of the row q_i and column q_i of M are stored. Checking whether a q_i was processed before can be checked locally on $f(q_i)$. Accessing the successors and predecessors of a node is as simple as in the matrix implementation. The load for the processors depends on the function f. Hence, we have similar advantages as in the case of the matrix implementation while avoiding that the maximal number of nodes has to be known in advance.

Actually, since first proposed [78, 23], this approach is widely used in the domain of parallel model checking [21]. Especially, Bell [9] studied the effect of function f with respect to equal distribution and number of processor crossing edges in greater detail.

3.4.2. Labelling the Game Graph

In the sequential setting, one would typically apply a depth-first search for labelling the nodes red and $green$, for example the one described by Lange [62]. However, it is not clear how to do this in parallel efficiently. Instead, we propose to extend the sequential algorithms described in Section 3.3.1 towards a parallel implementation. As explained in the previous subsection, it is easy to construct (a component of) the game graph in parallel employing a breadth-first search. When a terminal configuration is reached, a backwards coloring process can be initiated as described in Section 3.3.1. This can be carried out in parallel in the obvious manner. If all color information is propagated, the sequential algorithm performs a coloring of uncolored nodes and an erasing of light

run-time from a formal system description.

colors (cf. Section 3.3.1). It is no problem to do this recoloring on the distributed game graph in parallel. Through Theorem 3.3.7 or Theorem 3.3.10, no cycle detection is necessary but every workstation can do this recoloring step on its local part of the game graph.

However, to check that all color information has been propagated, a distributed termination algorithm is employed. Several algorithms for detecting termination have been proposed [73, 33]. As there is no dependency to the size of the game graph, their influence on our algorithms can be neglected, and thus they will not be discussed here.

Components may be labelled in a bottom-up or top-down manner as described in Section 3.3.1. Hence, we can characterize a family of algorithms parameterized on the order in which components are processed.

Theorem 3.4.1
The algorithms outlined in this subsection label a node (s, ψ) of the game graph $green$ if $\mathcal{T}, s \models \psi$. Otherwise, the node is labelled red.

3.4.3. A Family of Parallel Coloring Algorithms

To describe our approach in more detail, we proceed by developing several fragments of our algorithm in *pseudo code*. In particular, we show that the two steps of constructing the game graph and labelling the nodes can be carried out in an integrated way. More details about an actual implementation of the algorithms are given in Section 4.1.

Like in the sequential case, at the heart of our algorithm is the processing of a single game-graph component Q_j as depicted in Algorithm 3.5. Given a component (number), it expands all configurations of the component. It can be applied either for a parallel bottom-up or top-down labelling algorithm. As the color information of a terminal node is always correct (non-light) color, a coloring process is initiated, if a terminal configuration is reached. Colors are then propagated backwards.

Our algorithm is designed for a distributed setting. Each processor runs an unmodified copy, and we can only assume a local view of all data structures as explained in Section 3.4.1. Thus, we index the local part of a data structure with the number of its "owning" processor (index i for processor P_i).

For processors to communicate among each other, each P_i uses a queue $Work_i$ where processors can deposit requests, for example via some message passing mechanism. The algorithm then continually processes requests from its queue until the handling of the current component is completed. The locally known configurations of a game graph are stored in set $Conf_i$.

In lines 1–4, the component's initial configurations $\lceil Q_j \rceil$ are expanded, each via Algorithm 3.6. If a configuration $conf$ is not yet known, its successors $post(conf)$ are calculated and put on respective work queues. Then the algorithm enters a loop (lines 5–28), where it retrieves the next request msg, and processes it.

Algorithm 3.5 colorizeComponent$_i(Q_j)$, parallel version

Colorize those configurations of component $Q_j \cup \lfloor Q_j \rfloor$ owned by processor i.

```
1:  /* configuration conf from initial configurations of Q_j */
2:  for each conf ∈ ⌈Q_j⌉ do
3:    processSuccessors(conf, Q_j)
4:  end for
5:  repeat
6:    msg := get(Work_i)
7:    if msg = EXPAND(pred, conf) then
8:      if conf ∉ Conf_i then
9:        processSuccessors(conf, Q_j)
10:       initializeConfiguration(conf)
11:       λ(conf) := color(conf)
12:     end if
13:     if λ_i(conf) ≠ WHITE then
14:       put COLOR(pred, λ_i(conf)), Work_h(pred)
15:     end if
16:     →_i := →_i ∪ {(pred, conf)}
17:   else if msg = COLOR(conf, color) then
18:     decrement count(conf, color)    /* update color information */
19:     color' := color(conf)
20:     if color' ≠ λ_i(conf) then
21:       λ_i(conf) := color'
22:       for each pred ∈ pre_i(conf) ∩ Q_j do
23:             /* only work on current component */
24:         put COLOR(pred, λ_i(conf)), Work_h(pred)
25:       end for
26:     end if
27:   end if
28: until msg = COMPONENTCOMPLETED
```

Algorithm 3.6 processSuccessors$(conf, Q_j)$, parallel version

Process successors of configuration $conf$ in component Q_j.

```
1: Conf_i := Conf_i ∪ {conf}
2: if conf ∈ Q_j then
3:   for each s ∈ post(conf) do
4:     put EXPAND(conf, s), Work_h(s)
5:   end for
6: end if
```

Algorithm 3.7 color($conf$), parallel bottom-up version

Compute color for configuration $conf$.

```
1: if ∃color ∈ {RED, GREEN} : count(conf, color) = 0 then
2:     return color
3: else
4:     return λ_i(conf)
5: end if
```

Algorithm 3.8 recolorComponent$_i(Q_j)$, parallel bottom-up version

```
1: if Q_j is a μ-component then
2:     color := RED
3: else    /* Q_j is a ν-component */
4:     color := GREEN
5: end if
6: for each conf ∈ Q_j, λ_i(conf) = WHITE do
7:     λ_i(conf) := color
8: end for
```

Algorithm 3.9 Main procedure, parallel bottom-up version

```
for each component Q_j ∈ 𝒬 in bottom-up order do
    for each processor P_i in parallel do
        colorizeComponent_i(Q_j)
        recolorComponent_i(Q_j)
        Propagate colors from initial configurations ⌈Q_j⌉ to {Q | Q ⋖ Q_j}.
    end for
end for
```

In case of a request EXPAND$(pred, conf)$ (lines 7–16) to expand more of the game graph, we check whether the to-be-expanded configuration C has not yet been seen (line 8). It is then expanded (line 9) and initialized (line 10). A color label $\lambda(conf)$ is determined (line 11). It is then possibly propagated to predecessor $pred$ (lines 13–15). This request is put on the queue of the processor $P_{\mathrm{h}(pred)}$ who is responsible for configuration $pred$. A new game graph edge $(pred, conf)$ is then added (line 16). It is later needed to propagate color changes to predecessor configurations.

We process a coloring request COLOR$(conf, color)$ (lines 17–27) by recording that some successor of configuration $conf$ has just obtained color $color$ (line 18). Then, it is determined whether that color change has impact on $conf$ and its color is updated accordingly (lines 19–21). Also, on color update, the new color is propagated backwards to each predecessor $pre_i(conf) \cap Q_j$ of $conf$ in the current component (lines 22–25).

The processing continues until none of the processors has any requests left to handle, in which the algorithm finishes. This situation is detected by an termination check algorithm (not depicted here) which then inserts a message COMPONENTCOMPLETED into every processor's work queue.

Note how the coloring part (lines 17–27) of Algorithm 3.5 largely resembles its sequential counterpart Algorithm 3.3, as does the subroutine to determine a configuration's color (Algorithm 3.7). They mainly differ in using and updating only the locally known part of data structures.

Within the algorithm, some issues pointed out earlier are handled implicitly, and are worth mentioning.

Remark 3.4.2
The coloring of terminal configurations happens without special-casing, right after they are initialized (line 10). They are without successors, hence one of the counters (which one depends on the type of configuration) will be set to zero (Algorithm 3.1) causing them to be colored (Algorithm 3.7) immediately (line 11).

Note that never both RED and GREEN counters can equal zero, because of their initial values. Decrementing counters of a configuration $conf$ happens at most as many times as there are successors $post(conf)$ to that configuration, and the sum of both counters equals $1 + |post(conf)|$.

Only on two occasions do counters change in the algorithm (lines 10 and 18) and coloring requests are sent only if a color change occurs as a result of a change in counters. For the expansion case, it is sufficient to propagate the color to the predecessor $pred$ which caused the expansion (line 14). We can distinguish two cases:

- A configuration $conf$ is new $conf \notin Conf$, that is, only the single predecessor $pred$ is known. If $conf$ is a terminal configuration, its color is determined as lined out previously, and it is propagated to its $pred$.

- Configuration $conf$ has been visited before ($conf \in Conf$), thus we can assume that previous predecessors know its color already. Then, the color only needs to be propagated to the new predecessor $pred$.

On the other hand, in the coloring case a change of the color label must be propagated back to *all predecessor* configurations already known.

An important measure for distributed algorithms is their *message complexity*, that is the number of messages exchanged by all processors.

Lemma 3.4.3 (Component Message Complexity)
The number of requests for expansion and coloring a component is bounded by the size of the component. Thus, the message complexity of coloring a single component Q_j is linear with regard to its size $|Q_j|$.

Proof. For the case of expansion of the game graph, $\mathrm{processSuccessors}(conf, Q_j)$ is called exactly once per configuration (line 8–9). Then, for each edge $pred \to conf$ of the game graph, exactly one message $\textsc{Expand}(pred, conf)$ is sent.

The backwards color propagation process sends a single coloring request per edge on each color change. As a component's color changes at most once, one coloring request per edge is generated indeed. □

When coloring components in bottom-up order, each component is considered only once, the message complexity for Algorithm 3.9 is immediately clear from Lemma 3.4.3.

Theorem 3.4.4
The bottom-up coloring algorithm of game graph $\mathscr{G}$ has linear message complexity with regard to its size $|\mathscr{G}|$.

For the top-down coloring version (Algorithm 3.12) the same algorithm for coloring a component can be reused (Algorithm 3.5), provided the coloring of escape configurations (Algorithm 3.10) and the recoloring of uncolored configurations (Algorithm 3.11) are adjusted to deal with light colors.

3.4.4. Algorithmic Variations and Optimization Issues

We discussed several possibilities to process the components. In the examples shown in Section 3.3.1, we suggested a depth-first strategy. However, one could also use a breadth-first, bounded depth-first, or parallel breadth-first strategy. Depending on the employed strategy, the run-time of our algorithm is linear or quadratic with respect to the size of the game graph. Although the top-down case has a less appealing worst-case run-time, its improved on-the-fly behavior through the employment of light colors can

Algorithm 3.10 $\mathrm{color}(conf)$, parallel top-down version

Compute color for configuration $conf$

```
 1: color' := color(conf)
 2: if color' ≠ WHITE then
 3:    return color'
 4: else if Σ_{c∈{RED,LIGHTRED}} count(conf, c) = 0 then
 5:    return LIGHTRED
 6: else if Σ_{c∈{GREEN,LIGHTGREEN}} count(conf, c) = 0 then
 7:    return LIGHTGREEN
 8: else if conf ∈ ⌊Q_j⌋ then
 9:    if Q_j is a μ-component then
10:       return LIGHTGREEN
11:    else    /* Q_j is a ν-component */
12:       return LIGHTRED
13:    end if
14: else
15:    return λ_i(conf)
16: end if
```

Algorithm 3.11 $\mathrm{recolorComponent}_i(Q_j)$, parallel top-down version

```
 1: if Q_j is a μ-component then
 2:    color := RED
 3: else    /* Q_j is a ν-component */
 4:    color := GREEN
 5: end if
 6: for each conf ∈ Q_j ∪ ⌊Q_j⌋ do
 7:    if λ_i(conf) = WHITE then
 8:       λ_i(conf) := color
 9:    else if λ_i(conf) ∈ {LIGHTRED, LIGHTGREEN} then
10:       λ_i(conf) := WHITE
11:       count(conf, LIGHTRED) := 0
12:       count(conf, LIGHTGREEN) := 0
13:    end if
14: end for
```

Algorithm 3.12 Main procedure, parallel top-down version

```
for each component Q_j ∈ 𝒬 in top-down order do
  for each processor P_i in parallel do
    colorizeComponent_i(Q_j)
    recolorComponent_i(Q_j)
    for each components Q'_j ∈ {Q'_j | Q'_j ≺ Q_j} do
      Propagate colors from initial configurations ⌈Q_j⌉ to Q'_j
    end for
  end for
end for
```

be beneficial still, considering that a significant part of the game graph might not even be considered to obtain a result.

More optimizations are possible: Only the colors of escape configurations are needed when coloring a component. Thus, all other configurations of a child component can be deleted for coloring the current component. Furthermore, a configuration is of no use if all its predecessors are colored (with a full color). Especially in the top-down algorithm, this may be used to reduce the size of a component before the child components are considered.

In general, it is possible to design one variant of the algorithm in a way that only a single component is completely stored in the memory of workstation cluster while from the other components only the initial and escape configurations are needed. The price to pay for this implementation is that some components have to be reconstructed.

Another variant of our algorithm can be obtained by taking subformulas instead of the occurrence set of the subformulas for defining the graph of the formula (Definition 3.2.10). The resulting effect will be that the components of the game graph no longer constitute a tree order but form a directed acyclic graph. It can be expected that the resulting game graph is smaller since nodes may be shared. However, the tree order simplifies the decision when a component can be removed, as described in the previous paragraph.

Another optimization axis lies in the distribution function which decides where a configuration is stored. For example, the number of edges "crossing" different processors ($(s,\pi) \rightarrow (s',\pi')$ with $\mathrm{h}((s,\pi)) \neq \mathrm{h}((s',\pi'))$) should be reduced, as they trigger potentially expensive communication. Reconsidering that the state part s of a configuration can only change through unwinding a modality operator $[\![K]\!]\varphi$, this can be achieved to some extent by making the distribution function independent of the formula part π, thus coercing those configurations only differing in this part to the same processor.

3.4.5. Calculating Winning strategies

As pointed out already when motivating the games framework, a winning strategy does not only provide an answer to the model checking question but can also be applied for interactively debugging the underlying system. It is possible to extend our algorithm for not only asserting the existence of a winning strategy, but also providing one.

Note that the color of a terminal configuration in the game graph is a winning position for one of the players. If the color information is propagated to a predecessor without a color ($white$) and this leads to a color of the predecessor, it is clear how the corresponding winner has to chose. In other words, when a configuration becomes colored with $color$ due to some request COLOR$(pred, color)$ originating from the coloring of a configuration $conf$, the player's strategy is to choose $conf$ in configuration $pred$. If a configuration is colored during the recoloring phase (Algorithm 3.8 or 3.11, respectively), we pointed out in the proofs of Theorem 3.3.7 and Theorem 3.3.10 that the right strategy would be to choose a previously $white$ successor.

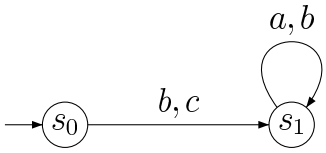


Figure 3.13.: A transition system.

3.5. Extensions towards L_μ^2

In this section we explain how model checking for L_μ^2-formulas can be carried out by iteratively using a procedure for checking L_μ^1-formulas. Firstly, we describe how to reduce alternation depth in the usual fixpoint computation. In the next subsection we show how this observation has to be read when dealing with game graphs. As result, we obtain a game-based sequential model checking algorithm for L_μ^2-formulas, which employs the game-based algorithm for L_μ^1-formulas as detailed in the previous sections. Finally, we describe in which way this algorithm can be parallelized.

3.5.1. Reducing Alternation Depth

The basic idea for reducing alternation depth is to resolve fixpoint variables of the maximal alternation depth using the iteration method. In the remaining computations only formulas with lower alternation depth are involved.

As a running example, we study the fixpoint computation for the transition system shown in Figure 3.13 and the formula $\mu X.[c]X \wedge (\nu Y.\langle a\rangle Y \vee \langle b\rangle X)$. The formula contains two variables X and Y of alternation depth 2 and 1, respectively. We compute the value of X in an iteration starting from $X = \varnothing$. The next approximation is given by

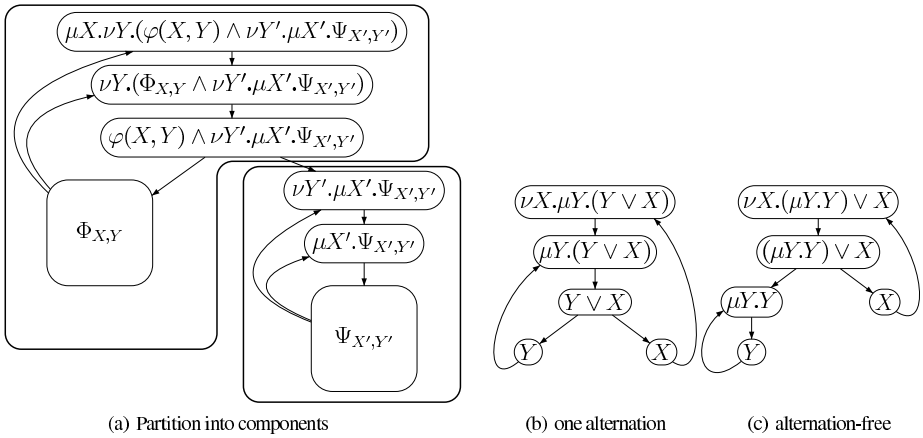
$$[\![\Phi_X]\!]_{V[X\leftarrow\varnothing]} = \{s \mid s \overset{c}{\not\rightarrow}\} \cap [\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\varnothing]}$$

The inner fixpoint formula (with value of X fixed to $\varnothing$) is alternation free. Hence, using our favorite L_μ^1 model-checking algorithm, we can find out that $[\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\varnothing]} = \{s_1\}$. This gives the next approximation of $X = \{s_1\}$. In the following iteration we compute

$$[\![\Phi_X]\!]_{V[X\leftarrow\{s_1\}]} = \{s_0, s_1\} \cap [\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\{s_1\}]}$$

and again use an L_μ^1 model checker to find $[\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\{s_1\}]} = \{s_0, s_1\}$. This ends the computation since a further iteration cannot add any new states to the value of X.

In general, the situation can be more complex than in our example. For instance, several variables of maximal alternation depth can occur in a formula, possibly with



(a) Partition into components (b) one alternation (c) alternation-free

Figure 3.14.: Graphs of formulae

different types. Variables of the same type can depend on each other. Also, if $Y \sqsubseteq X$ then we must know the value of Y before we start to compute X. Hence evaluation of these variables must be performed in bottom-up manner, using simultaneous fixpoint computations.

More precisely, let $\mathcal{X} \subseteq Var(\varphi)$ be the set of variables with maximal alternation depth. Systematically we evaluate them using the iteration method. We start with these variables in $\mathcal{X}$ which do not subsume any other variables from $\mathcal{X}$. After finding their values we proceed to the variables $X \in \mathcal{X}$ for which values of all $Y \sqsubseteq X$, $Y \in \mathcal{X}$ are already known. Eventually all variables in $\mathcal{X}$ will be evaluated.

In each stage of this process we find the values of all μ-variables and, separately, the values of all ν-variables by simultaneous iteration. Specifically, if $X_1, \ldots, X_p$ are all μ-variables which we want to evaluate, then we start by setting $V(X_i) = \varnothing$ for $i = 1, \ldots, p$, where V is the current valuation (keeping the already computed values). Then in each iteration values of $X_1, \ldots, X_p$ are updated using formula $V'(X_i) = [\![\Phi_i]\!]_V$. Here Φ_i is the formula Φ_{X_i} with all subformulas $\sigma Y.\Phi_Y$, $Y \in \mathcal{X}$ replaced by Y. Note that if $Y \sqsubseteq X_i$ for $Y \in \mathcal{X}$ then we have already computed the value of Y and it is stored in V. Observe that the alternation depth $\mathrm{ad}(\Phi_i)$ of each Φ_i is strictly lower than the alternation depth of the original formula $\mathrm{ad}(\varphi)$.

3.5.2. Alternation and Game Graphs

The previous idea of reducing alternation depth can be applied similarly for game graphs. To see this, we study the structure of game graphs for L^2_μ-formulas.

As explained in Section 3.2.2, an arbitrary graph can be partitioned into maximal strongly connected components and directed acyclic graphs. Furthermore, these components can be partially ordered by bridges (Figure 3.14(a)). For the graph of an

L^2_μ-formula, we can easily see that within its strongly connected components either a μ-fixpoint or ν-fixpoint is unwinded (Figure 3.14(c)), or one alternation of μ- and ν-formulae (Figure 3.14(b)).[6] We conclude:

Theorem 3.5.1
Let $\varphi \in L^2_\mu$. Then there exists a partition of $\mathcal{G}(\varphi)$ such that every subgraph either is a directed acyclic graph, contains only variables of alternation depth one, or contains variables of alternation depth one and two which are dependent. Furthermore, the subgraphs are partially ordered by bridges.

In Figure 3.14(a), we indicate the partition of the graph of formula $\mu X.\nu Y.(\Phi_{X,Y} \wedge \nu Y'.\mu X'.\Psi_{X',Y'})$. Here, $\Phi_{X,Y}$ and $\Psi_{X,Y}$ are arbitrary fixpoint-free formulas with free variables among X, Y and X', Y', respectively. Note that X subsumes Y' but Y' is not dependent on X.

Since the game graph can be seen as *unfolding* of the formula graph using the transition system, the partition of the formula graph induces a partition of the game graph. Together with Theorem 3.5.1 we get:

Theorem 3.5.2
Let $\mathcal{G}$ be the game graph for a transition system and an L^2_μ-formula. Then there exists a partition of $\mathcal{G}$ such that every subgraph is either a directed acyclic graph, contains only variables of alternation depth one, or variables of alternation depth two and one that are dependent. Furthermore, the subgraphs are partially ordered by bridges.

3.5.3. Coloring Algorithm for L^2_μ

We now develop the algorithm for showing the existence of a winning strategy for one player in L^2_μ games, and hence solving the model checking problem for L^2_μ. The algorithm preprocesses the given game graph component-wise (using a partition according to Theorem 3.5.2). It employs the sequential algorithm for the alternation-free fragment L^1_μ of the μ-calculus presented in Section 3.3.1 to color each component, possibly repeatedly with further intermediate processing steps.

The basic idea of the algorithm is to explicitly perform any fixed point calculation with alternation depth 2 directly, then reusing the algorithm for L^1_μ to perform the remaining inner ($ad(X) = 1$) fixpoint calculations.

Preprocessing the Game Graph. First, we decompose the given game graph according to Theorem 3.5.2 into partially ordered components Q_i, which are subsequently colored, starting with the least ones.

[6] Note that the latter case does not occur when formulas from L^1_μ are considered.

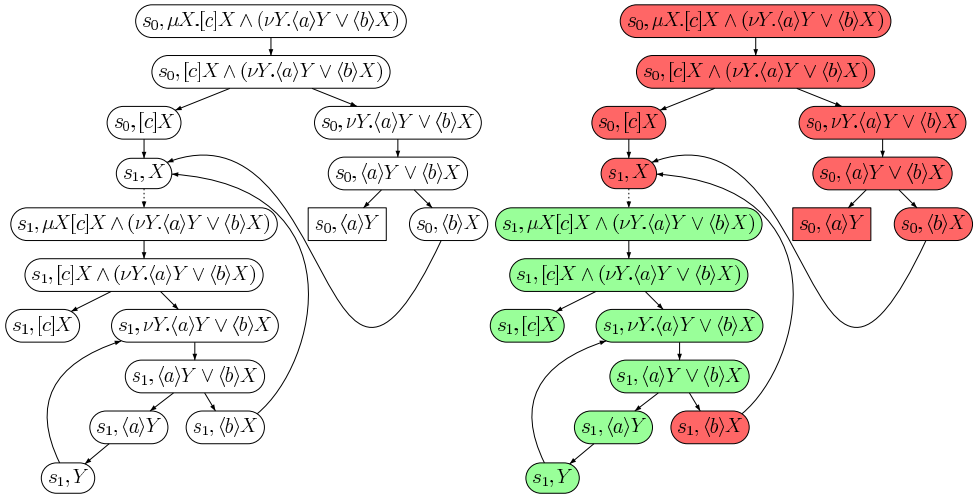


Figure 3.15.: Coloring a game-graph component with alternation.

Let $G_i = (Q_i, E_i)$ be the graph of a single component. If G_i is a directed acyclic graph or contains only variables of alternation depth one, Algorithm 3.3 can be used immediately on Q_i. Assume now that G_i contains a variable X of alternation depth $\mathrm{ad}(X) = 2$. To simplify the presentation, we assume X to be a μ-variable. The forthcoming explanation can as usual be dualized for ν-variables.

We determine the set $PE_i \subseteq E_i$ of edges $(s, X) \to (s, \mu X.\Phi_X) \in E_i$ (for $ad(X) = 2$). The modified graph $G_i^0 = (Q_i, E_i \setminus PE_i)$ does not admit paths containing infinitely many configurations of *both* types of fixpoint formulas any more. Figure 3.15(a) shows the game graph for our example. The dotted edge is the one removed to obtain a game graph in which no variable of alternation depth 2 is unwinded.

Coloring the Game Graph. Next, we apply a pre-coloring of configurations q (which are now leaves in the modified graph) of edges $q \to q' \in PE_i$ according to the type of their corresponding fixpoint formula—*red* for μ fixpoints, *green* for ν. Then we execute Algorithm 3.3 on the pre-colored and modified G_i^j (initially $j = 0$), which is afterwards completely colored.

A new uncolored graph G_i^{j+1} is created from G_i^j, and for each edge $q \to q' \in PE_i$, we copy the color of q' in G_i^j to configuration q in G_i^{j+1}. They form the refined assumptions of the pre-coloring. Algorithm 3.3 is executed again on G_i^{j+1} afterwards.

The above step is repeated until $G_i^{j+1} = G_i^j$ and hence the fixpoint is reached. The number of iterations is bounded by $j \leq |proj_1(PE_i)|$.

In our example, the configuration (s_1, X) is initialized with red. Algorithm 3.3 will color the game graph as depicted in Figure 3.15(b), in which red configurations are filled

Algorithm 3.13 L^2_μ-colorizeComponent(Q_j)

```
1: Remove edges PE_j from component graph of Q_j
2: while exists pred → conf ∈ PE_j with color mismatch λ(pred) ≠ λ(conf) do
3:    for each conf ∈ Q_j do
4:       if pred → conf ∈ PE_j then
5:          λ(pred) := λ(conf)
6:       else
7:          λ(conf) := WHITE
8:       end if
9:    end for
10:   colorizeComponent(Q_j)    /* invoke L^1_μ-algorithm */
11: end while
12: Add edges PE_j back to component graph of Q_j
```

with ■ and *green* ones with ■. As the color of configuration $(s_1, \mu X[c]X \wedge (\nu Y.\langle a\rangle Y \vee \langle b\rangle X)))$ is different from (s_1, X), we set the color of configuration (s_1, X) to *green* and start a new iteration. Now, Algorithm 3.3 colors all configurations green (except $(s_0, \langle a\rangle Y)$). The color of $(s_1, \mu X[c]X \wedge (\nu Y.\langle a\rangle Y \vee \langle b\rangle X))$ has not been changed in this iteration, indicating that we reached a fixpoint, and we can stop the coloring at this point.

Proceeding with the remaining components is done in the same way, the propagation of colors *between* components is the same as described before. The result of the extended algorithm for L^2_μ is a completely and correctly (conforming to the rules lined out in Section 3.2.4) colored game graph.

The pre-coloring preserves the property of Algorithm 3.3 that the coloring is monotonic, that is no configuration changes its color more than once. As Algorithm 3.3 terminates, the extended version is terminating as well. We sum up:

Theorem 3.5.3
Given a transition system and an L^2_μ-formula, the extended algorithm for L^2_μ constructs the corresponding game graph and labels the configurations either *red* or *green*, depending on whether ∀belard or respectively ∃loise has a winning strategy.

Complexity The run-time of our coloring algorithm for L^2_μ is quadratic in both the number of states s of the underlying transition system, and in the length l of the given L^2_μ-formula, thus $s^2 \times l^2$. This is straight-forward to see, recalling that the complexity of Algorithm 3.3 is bounded by $s \times l$ as stated in Section 3.3.1, and Algorithm 3.3 is executed at most $|\bigcup_i proj_1(PE_i)| \leq s \times l$ times.

The costs for generating the PE_i are negligible here, since it needs to be done only once (for example, while producing the game graph), and is linear with respect to the number of edges in $\mathcal{G}$.

4. Implementation and Empirical Results

The algorithms we have proposed so far have been implemented in a system called *UppDMC* by Holmén et al. [52]. In this section, we will summarize their efforts. First, we give an overview over the structure of the implementation, and then proceed by conducting independent measurements with the same setup, confirming their results and drawing further conclusions.

4.1. The UppDMC Implementation

The UppDMC system is developed in C++ using the message passing standard MPI [42] for communication among the different computers. The new implementation does not depend on our previous implementations in Haskell and C++ [13] and is more focused on performance.

While the algorithms outlined in the previous sections are designed to be carried out *on-the-fly*, the current version of UppDMC only makes partial use of it. In particular, for the measurements shown in the next section, the algorithms work on previously generated transition systems. This is only due to practical reasons: To be able to compare UppDMC with existing model checkers, they used precomputed transition systems made available as the *Very Large Transition System* (VLTS) benchmark suite.[1] It would however be possible to lift this restriction and adapt UppDMC to compute transition systems on-the-fly as proposed in our algorithms.

Figure 4.1 gives an overview of the general structure of UppDMC. The implementation is divided into several modules.

μ-Calculus Module. The μ-calculus module is responsible for parsing formulas, building formula graphs, and partitioning them into components as proposed in Section 3.2.2. These graphs are then used to construct the game graph. Also, the alternation depths $\mathrm{ad}(\cdot)$ for fixpoint variables contained in a formula are determined.

Transition-system Module. In the transition system module, states and transitions are read from pre-generated files into memory. For each processor P_j, only those states s are stored which are assigned to it by the distribution function $\mathrm{h}((s, \pi))$ (Section 3.4.1).

[1] `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`

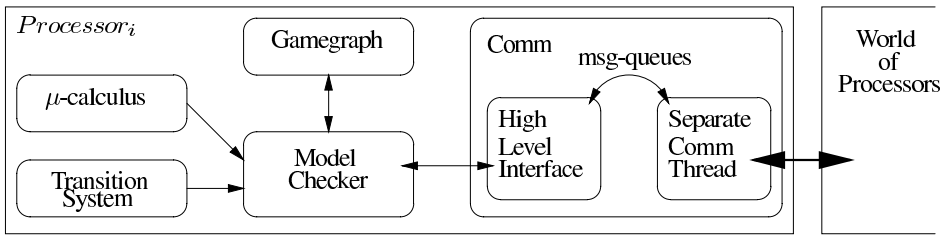


Figure 4.1.: Simplified structure of UppDMC.

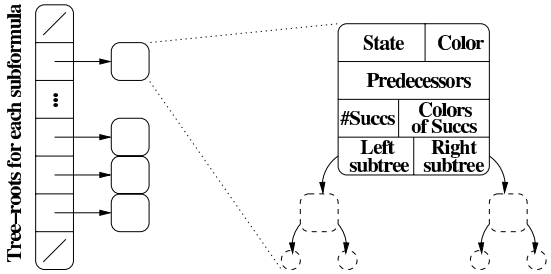


Figure 4.2.: Game-graph data structure.

The module provides search functions to access all successors if provided a state and action label. The authors report memory requirements for storing transitions of pairs consisting of a predecessor and label as $12 + 4 \times t$ bytes, where t is the number of transitions.

Game-graph Module. The model checker's main data structure is defined in the game-graph module. To conserve memory, game-graph configurations (s, π) are grouped by their formula part π, which also allows for efficiently retrieval of initial configurations $\lceil Q_j \rceil$ of a component Q_j. This operation is part of the whole family of algorithms we proposed previously, as they work component-wise.

Another memory conserving means described by Holmén et al. is the compression of predecessor configurations. In most cases, their formula part can be deduced from the statically calculated formula graph, and thus need not be saved explicitly. Furthermore, configurations with only one successor are not stored, as their color is immediately clear from this single successor. Initial nodes and alternating variables however are never removed.

The data structure for storing a game-graph and their configurations is outlined in Figure 4.2. Each game-graph configuration is reported to occupy about 36 bytes of memory plus the data needed to store the predecessor of the node. The authors report

that a cluster with total memory of 1 TeraByte could check a transition system of size $2*10^9$ states for livelock (for example, from Figure 4.3). This is based on the assumption that the transition system has an average branching factor of 10, which was empirically derived from real-world data points from on the VLTS suite.

Communication Module. The communication module provides a high level interface for processors to communicate via sending and receiving messages. It is used in both the algorithm and the termination-detection module. Choices for the transport layer are either TCP/IP sockets or MPI [42]. Communication queues are decoupled from the actual algorithm via a threaded implementation, and buffering and timeouts are supported for efficient utilization of the network.

Termination-detection Module. The termination detection module is used to determine when a phase of the algorithm (for example, expansion of a component) ends globally, and the next one (for example, recoloring) can start.

Implementation of Algorithms. The model-checker module implements a template with a formula object, a transition system object and communication object as parameters. The implementation closely resembles the algorithms we developed in the previous sections.

4.2. Practical Experiences

The experiments of Holmén at al., our replications of them and our additional measurements have been conducted on the Aachen University *Parsecs Cluster*. This Linux cluster consists of 26 machines, each equipped with two 500 MHz Intel Pentium III and 512 MB main memory (of which around 400 MB are accessible for us). The machines are interconnected with a private 100 MBit switched Ethernet network. We were able to reserve the cluster for exclusive benchmarking. No other significant computation tasks were running during that time.

Each machine runs two threads, one to handle network communication and the other for the actual algorithm. As both parts can run independently from each other, we get a slight performance advantage through overlapping I/O and calculations.

For measurements, we used transition systems from the VLTS benchmark suite, and thus we were constrained to check properties which are compatible with them. Figure 4.3 shows two of the formulas we tested with, *NoDeadlock* and *Livelock*.

In Figure 4.4, we present the run-times of of our tests. We recorded the wall-clock time for the best of three runs for each test. The measurement methodology with which

Property	Formula
NoDeadlock	$\nu X.([-]X \wedge \langle - \rangle \texttt{true})$
Livelock	$\mu X.(\langle - \rangle X \vee \nu Y.\langle \tau \rangle Y)$

Figure 4.3.: Two μ-calculus formulas used during testing.

VLTS Name	# of States	# of Transitions	Checked Formula	
			NoDeadlock	Livelock
vasy_2581_11442	2,581,374	11,442,382	44 sec	47 sec
vasy_4220_13944	4,220,790	13,944,372	56 sec	67 sec
vasy_4338_15666	4,338,672	15,666,588	64 sec	64 sec
vasy_6020_19353	6,020,550	19,353,474	59 sec	125 sec
vasy_6120_11031	6,120,718	11,031,292	95 sec	108 sec
cwi_7838_59101	7,838,608	59,101,007	149 sec	314 sec
vasy_8082_42933	8,082,905	42,933,110	162 sec	134 sec
vasy_11026_24660	11,026,932	24,660,513	150 sec	160 sec
vasy_12323_27667	12,323,703	27,667,803	160 sec	177 sec
cwi_33949_165318	33,949,609	165,318,222	560 sec	8715 sec

Figure 4.4.: Elapsed time in seconds for checking ten of the largest transition systems of the VLTS benchmark suite with 25 machines. Formulas checked were *NoDeadlock* and *Livelock*.

we conducted our experiments and will present results is based on guidelines proposed by Crowl [30], in order to ensure accuracy and effectiveness of our presentation.

To our satisfaction, we were able to almost exactly reproduce the run-times reported by Holmen et al., as we used an identical setup. For all except one of the tested systems, we obtained results within 6 minutes, often in less. The exception is a transition system named "cwi_33949_165318", which with roughly 33.9 Million states is the biggest system we tested. Handling of this system exceeds the total memory of the cluster, resulting in a ten-fold increase in run-time due to heavy swapping to disk, as the memory fills up. However, despite the run-time penalty we were able to obtain a result.

In Figure 4.5 we can see the dramatic effect of increasing the number of machines on the run-time. Doubling the number of processors roughly reduces the amount of time needed to obtain a result by a factor of two, showing the predicted linear speedup of our algorithm. A better view on the scalability of our algorithm can be given by consider-

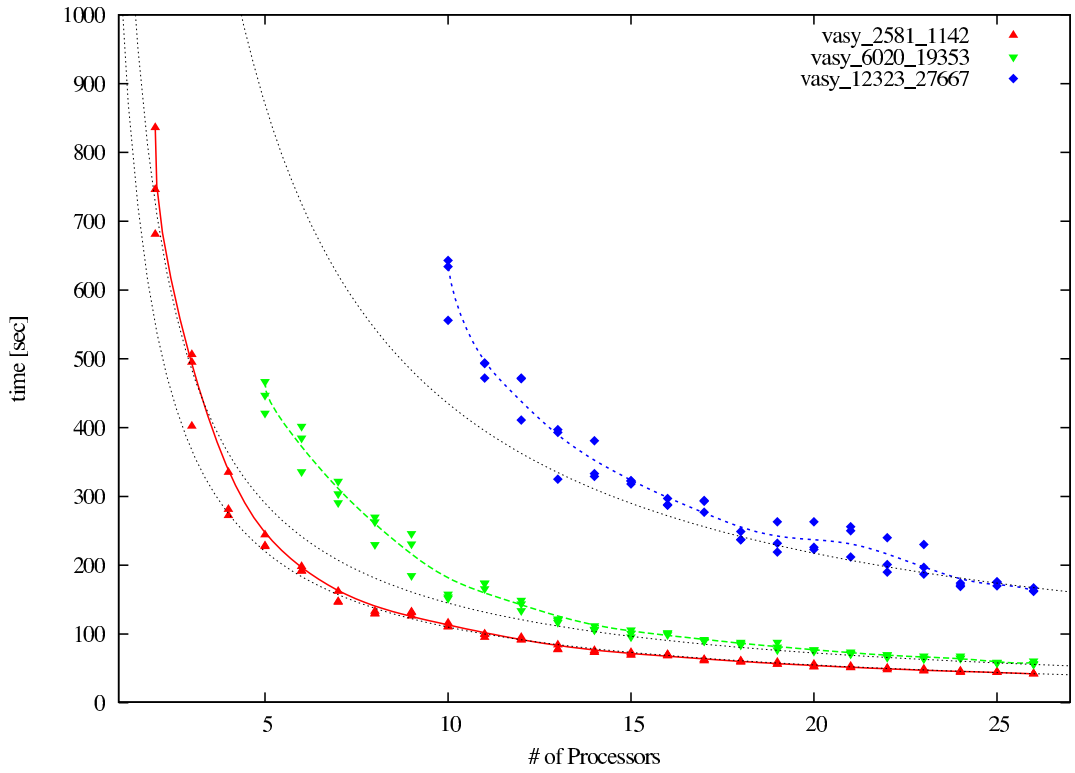


Figure 4.5.: Absolute run-times (in seconds) for checking the *NoDeadlock* formula on three VLTS transition systems on up to 26 processors. We measured three runs for every system, colored lines are interpolation of these values. Dotted black curves denote asymptotic run-time based on the value for 25 processors.

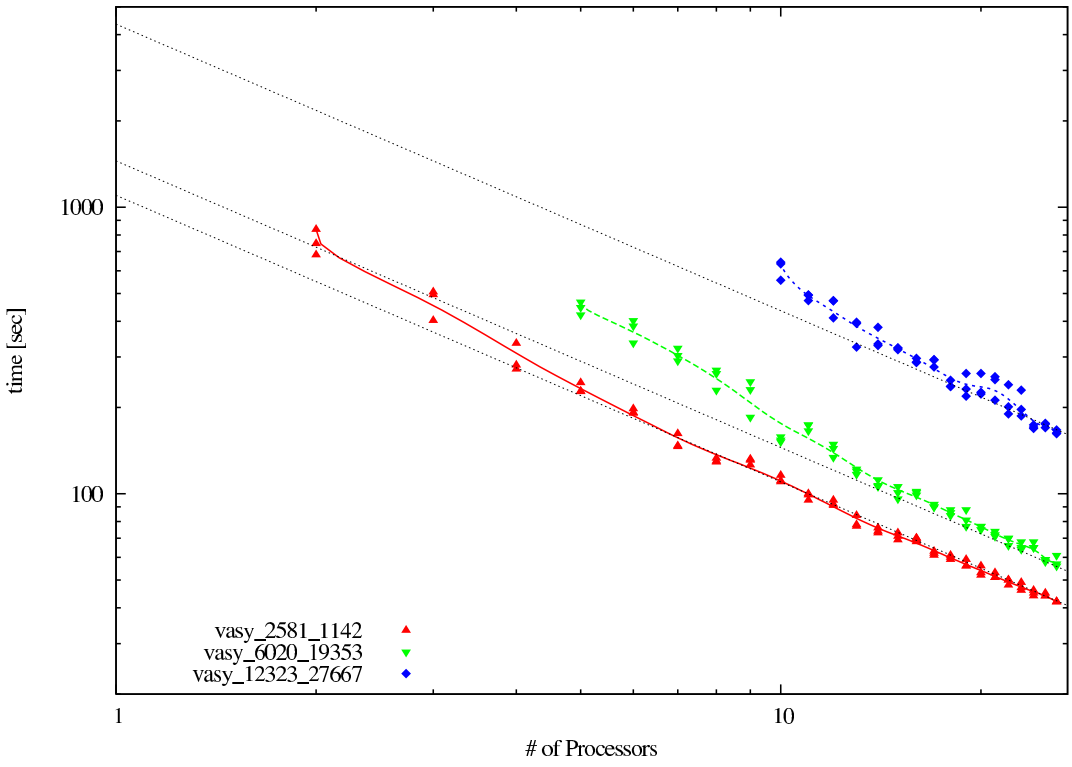


Figure 4.6.: Absolute run-times (in seconds) for checking the *NoDeadlock* formula on several VLTS transition systems on up to 26 processors. Both scales are logarithmic, revealing linear speedups. Dotted black curves denote asymptotic run-times based on the value for 25 processors.

ing measured data points in log-log-scale (Figure 4.6), which abstracts from absolute performance and is independent from processor technology used in our experiments.

For each transition system, the dotted black lines denote estimated *asymptotic run-times* based on the value for $N = 25$ processors in our case. If our algorithm takes t_N seconds to compute a result for transition system $\mathcal{T}$ on N processors, we estimate the asymptotic run-time curve for this setting as function

$$f_{\mathcal{T}}(p) = t_N \times \frac{N}{p} \text{ [sec]}$$

with p representing the actual number of processors. We can observe that especially with less processors the measured times are larger than predicted by $f_{\mathcal{T}}(p)$.

Limiting Factors. Our investigation revealed two responsible factors:

- Memory consumption on individual machines exceeds the physical amount of memory. Also, as memory pressure lowers, run-times become less scattered.
- Data per machine increases, penalizing data structure insertions and lookups with higher costs. If data can be distributed to more machines, both operations become cheaper.

  Another factor with similar effects could be improper distribution of game-graph configurations. However, we were able to rule this out. Measurements of the quality of the implemented distribution function showed its adequacy (Figure 4.7). Configurations are evenly distributed, with little deviation from average on all our tests.

Another limiting factor of our algorithms is the structure of transition systems. A system with low branching factor does not permit a lot of parallel computations within our algorithms, hence degrading effectiveness of a distributed setup.

On Relative Speedups. Note that in our discussion of results, we consciously avoided *relative speedup* diagrams as they are often misleading. We are usually unable to obtain a result for the canonical normalization point—a single processor ($p = 1$), or better yet, a sequential version of the program—because a computation exceeds time limits or available memory. A suitable base case can perhaps be interpolated, however this must be done with a great deal of care, as choosing a conservative value would present results in a too optimistic light.

In contrast, we based asymptotic approximations (dotted black curves) on the *best values* that we obtained in our tests, thus only pessimizing our results. Also, we chose to base our interpretations on absolute values of our measurements, which speak for themselves, and can easily be compared to.

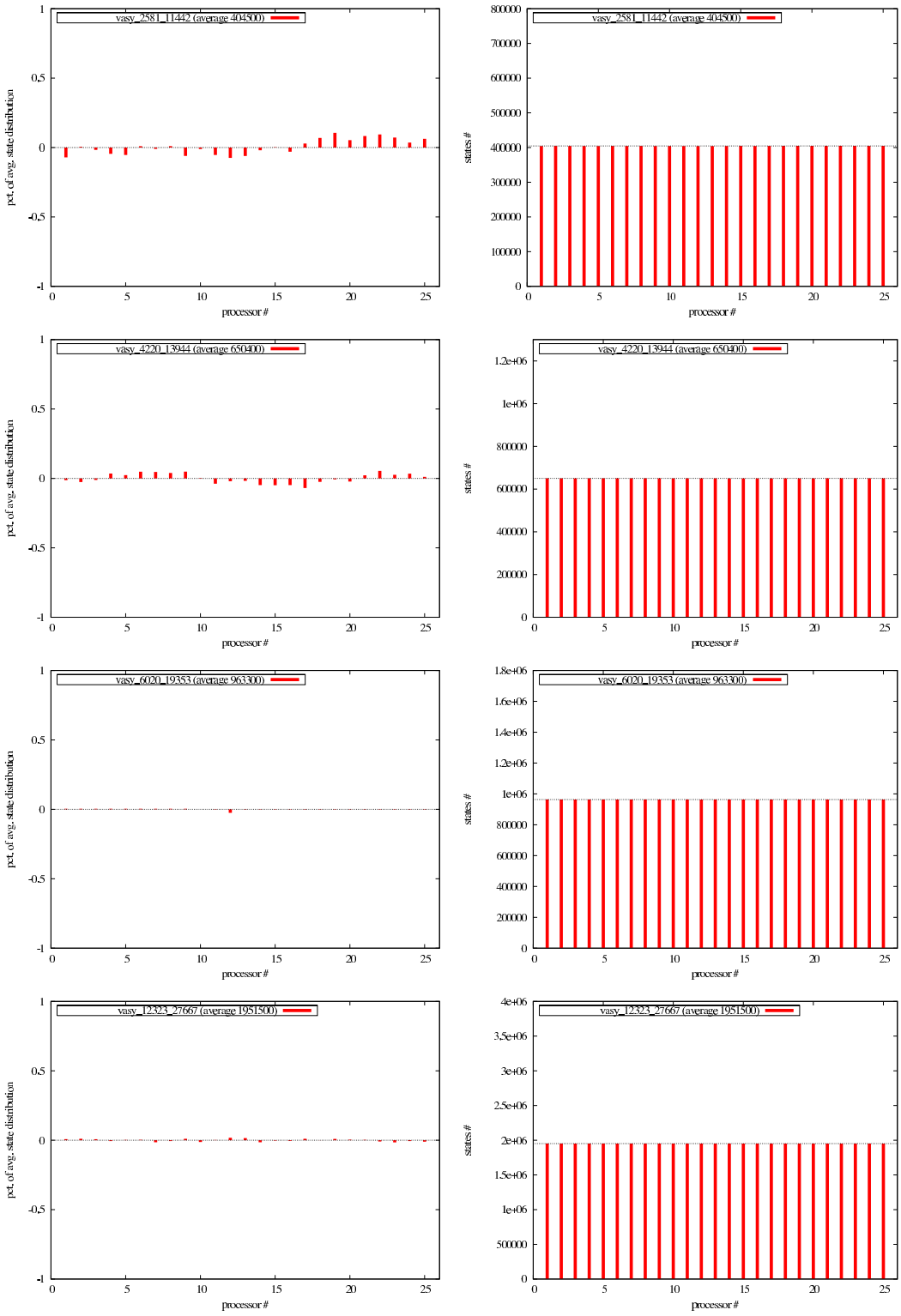


Figure 4.7.: Relative and absolute distribution of states among processors. All experiments exhibit even distribution, with little deviation.

Transition system	Property
	Livelock
vasy_8082_42933	Not satisfied
vasy_11026_24660	Not satisfied
vasy_12323_27667	Not satisfied
cwi_33949_165318	Satisfied

Figure 4.8.: Previously unknown results from the VLTS suite.

Practical Relevance. Our experiments show that the algorithms we developed behave well on real-world data and display the scalability we expected. Additionally, it is worth stressing that for most of the larger examples in the VLTS benchmark suite it was previously not possible to check for live-locks. With the UppDMC implementation of our algorithms, we were able to fill in all missing results (Figure 4.8 gives an overview), and even within reasonable time, on a cluster that would be considered small by today's standards.

Part II.

State Space Generation

5. State Space Generation

5.1. Introduction

In the previous part we developed distributed algorithms to solve the model checking problem for fragments of the μ-calculus. We were able to show their effectiveness by benchmarking them on the *Very Large Transition System* (VLTS) benchmark suite, a set of pre-generated *state spaces* from real-world models.

However, generating state spaces from a concise description takes up considerable amounts of time. Our model checking algorithms were designed with this in mind already. Both, generation and actual checking of properties can be carried out simultaneously and even profitably. As our algorithms work *on-the-fly*, only in the worst case we need to generate the whole state space. Depending on the property to check, our algorithms are able to prune large parts of the state space which are not significant for the outcome of the verification run. While this already reduces run-time simply by doing less work, it can even help in cases where it is impossible to store the full state space because it is prohibitively large.

Our goal is now to develop an efficient and reuseable method to generate state spaces from high-level descriptions, which is suitable for integration with our algorithms.

5.2. Status Quo

Common approaches in state-based model checking employ modeling languages like CSP [51], LOTOS [16], Murφ [34], DVE [4], or PROMELA [55] to describe actual state spaces. These languages are usually non-trivial: in addition to concepts found in programming languages (scopes, variables, expressions, etc.) they often provide features like a process abstraction, non-determinism, guarded commands, synchronization and communication primitives, timers, etc. Implementing an operational model of such languages for use in verification tools is consequently not straightforward, even more so if the language is described informally only, and their static and operational semantics are incomplete at best, outdated, or entirely unavailable.

That being said, when developing verification tools it is highly desirable to reuse an already existing popular modeling language like e.g. PROMELA, which has been used in a sizeable number of real-world case studies. These models can be used to benchmark new tools against old ones in a fair way. PROMELA has wide industry acceptance, al-

lowing modelers to try out compatible tools without having to re-specify models in yet another formalism. Lastly, since these tools are developed *for the application* of formal methods, it is worthwhile and only fair to treat them with the same rigor: a shared underlying virtual machine would make it possible to compare different algorithms fairly and easier to test a new algorithm for conformance against existing algorithms.

This begs however the question, why existing tools are not simply extended and thus the whole static and dynamic semantics machinery is reused? The most trivial answer is that new approaches might be implemented in a different programming language, for a variety of reasons [67]. Also, many verification tools can be considered research prototypes which are not developed with extensibility first in mind: modifications become more time-consuming. Furthermore, different approaches often admit different tool architectures: parallel and distributed model checkers often need small *mobile data structures*, so that computations can be relocated to other processors. In particular, the algorithms we have developed previously fall into this category.

Nevertheless, as our first proposition we stress that despite different designs all these tools have in common the need for a state space generator component.

Another commonly found reuse pattern is the translation of other formalisms to modeling languages like PROMELA [79, 36], using SPIN [48] as verification back-end, and hence restricting the choice of analyses to offerings of a single tool again, or getting trapped in *abstraction inversion*, that is, non-trivial encodings of constructs [2].

It comes as no surprise that researchers often find it easier to invent their own modeling language with informally specified semantics incompatible to existing tools which, as argued above, puts additional burden on end-users to switch tools, benchmark them, or consider them at all.

In order to remedy current shortcomings we propose a virtual machine-based approach to state space generation, in which high-level modeling languages are first translated to an intermediate format consisting of byte-code instructions. Subsequent execution of such byte-code programs with a virtual machine can be very efficient and yields state spaces for further use in model checking tools.

5.3. Contributions

In order to base our work on a solid foundation we present a formal model for our virtual machine. The operational semantics of our virtual machine are straight-forward, and hence easy to derive an implementation from. Moreover, our machine model can be augmented to handle timers, probabilities etc., in a compositional way by adding instructions, and keeping the rest of the model unchanged.

Most byte-codes are simple operations, and benchmarks show that our machine is competitive in state space generation to SPIN. For distributed state space generation our

machine offers additional benefits: it can be restarted from machine state snapshots, which have a self-contained, contiguous and platform-independent representation, and thus can be send across networks to without further serialization efforts.

As further contribution, by translating PROMELA to our byte-code language, we get up-to-date *executable* operational semantics for PROMELA essentially for free. Previous attempts of PROMELA semantics have been found to be inadequate for our purposes (Section 7.6.1). The translation also allows us to apply conventional compiler techniques such as code optimization, control and data-flow analyses. We stress however that our machine is generic and not exclusively tied to PROMELA.

5.4. Overview

In the next chapter, we review desirable requirements for an intermediate language. In section 7.1 we describe the virtual machine model and byte-code semantics. Section 7.2 gives an example how the virtual machine can be used for state space generation. In section 7.3 we report on a use case for our virtual machine: the translation of PROMELA to our byte-code language. Section 7.4 presents benchmark results for our implementation. We conclude with a brief summary of related work in sections 7.6 and 7.7.

6. Intermediate Formats

In formal verification, the goal is to analyze a model with respect to given properties. We described algorithms in Section 3.4 which require as input a model given in terms of a (possibly labelled) transition system. We can see the transition system as the language describing all possible behaviors of a model. The basic concepts in this language are states and labelled transitions, that is, relations between states. However, it is quite tedious for a model designer to directly use this formalism for the specification of anything but small models, as transition systems merely allow a verbose and *flat view* on the model. Within the transition system, there is no mechanism to manage the complexity of a model through hierarchical decomposition.

Thus, for convenience of the designer, usually a *high-level language* is offered to compose models in a concise and hierarchical way. Such a language provides some level of abstraction above the type of model understood by verification algorithms, but also opens a semantic gap. This gap can be bridged by a *translation procedure* which expresses the semantics of the high-level language in terms of the *low-level* language used by algorithms—states and transitions.

Experience shows that this translation is an involved process, as high-level modelling languages offer advanced constructs to easily specify systems on a very abstract level. Besides notions found in typical programming languages, we can identify some features common in many modelling languages:

Non-determinism From the current point in time, more than one alternative future of a computation is possible. Verification tools commonly take all of them into account.

Concurrency *Process abstractions* allow separate modeling of related subtasks and thus act as structuring device. Additionally, they concisely describe independent behavior.

Synchronization For models of *dependant* behavior, processes can influence each other through means of semaphores, (a)synchronous communication channels, and global state changes.

Priorities A notion of preference of one action over another allows to specify richer models which closer resemble reality. This includes prioritized actions as well as simple notions of time.

Note that some of these features may be expressed in terms of others, but this may lead to less intuitive ways to model the underlying system. Therefore, we consider all of these features to be present in a typical language.

The translation process which expresses these features in terms of a low-level description amenable for verification has to fulfill several desirable requirements. We highlight the one's we consider most important:

T1 The translation should be *automatic*.

T2 It should be *time* as well as *space efficient*. The translation procedure should be finished quickly, as the important task is the verification of the model, not its translation into another format. Also the output of the translation must still be reasonable to store it.

T3 A low-level model must capture the meaning of the high-level model, thus a translation should be *semantics-preserving*.

T4 It must be *traceable*. Principally, model designers deal with high-level descriptions. If we now consider a translation from some language high-level language L to a lower-level language L', the results of a verification run are too expressed in terms of L'. This makes presentation of counter examples non-intuitive[1]. Thus, any translation must be *reversible* or *traceable* at least to some extent so that results can still be presented relative to the user's input language L.

In general, we could either translate a high-level directly into a low-level representation, or use an intermediate step which simplifies the translation process and can also be reused as common ground from other high-level languages. Optimizations and further transformations can then operate on this intermediate format.

6.1. Direct Translation

One possible translation approach is to interpret the high-level description directly to obtain possible successor states, when the system is in a given state. Interpreters of this kind usually facilitate a rewriting strategy which assigns small-step semantics to the control structures and data manipulation of a language.

As interpreters work directly on the high-level specification, it is easy to obtain direct matches between steps in the abstract and steps in the concrete system (T4). However, this approach is usually neither time nor space efficient (T2). Furthermore, the effort

[1] For analogy, consider a compiler for a regular programming language which presents compile-time errors of a program on the level of its internal tree representation instead of source code locations.

needed to develop and change a state-space generator is tedious when done manually, since there is not much opportunity to share building blocks.

Development efforts can be simplified by either using configurable *rewrite engines* (such as ELAN [17] or Maude [27]) or dedicated *specification language compilers*. The idea of the latter is that the syntax and semantics of the underlying specification formalism is formalized. Then, a tool can generate a corresponding parser and small-step semantics functions automatically. Two such systems have been developed as proof of concept, PAC [28] and SLC [66].

The Process Algebra Compiler (PAC) takes a process algebra description and generates a front-end for the NCSU Concurrency workbench. The description consists of a specification for the syntax and semantics, given in form of *structural operational semantics* (SOS) rules. The *specification language compiler* (SLC) follows the same idea, but employs Rewriting Logic [75] as a framework rather that SOS rules, which has some advantages when formalizing semantics.

These prototypes show that the approach works well, especially when engineering new specification language formalisms. The corresponding syntax and semantics can be adapted easily, always yielding a complete verification environment for each version of the formalism studied.

However, the drawback of the approach is the lack of efficiency. Either the automatically generated interpreter lacks performance, as in the case of SLC, or non-obvious manual optimizations are needed as in the case of PAC.

Turning to rewrite engines, we face similar problems. Although tools like ELAN use powerful compilation techniques of rewrite rules, the overall efficiency of the resulting system is poor, as reported by Leucker and Noll [66]. The most important reason is an interfacing problem. Marshalling of data structures used in the verification tool to those used in the rewriting engine slows down the overall performance and yields an uncompetitive approach [14]. However, if a rewrite engine is provided in form of libraries which can be linked to the final system and compatible data structures are used, they might be an option. Only recently, such a library version of ELAN has become available.

6.2. Using an Intermediate Format

Inserting intermediate steps into a translation from high to low-level descriptions is a well-established technique in compiler construction (*compilation by transformation*). For modeling languages this approach was proposed before (most recently by Bozga et al.[18] and Garavel et al.[43]), and found to be helpful. We briefly enumerate important properties of a well-designed intermediate language as put forward by Garavel et al. [43] for the example of (E-)LOTOS. For a detailed discussion we refer the reader to that

paper.

The authors argue that a suitable intermediate language should aim at providing a simpler semantic model, and in particular it should support

I1 Conditions on input variables, e. g. reception of value v from channel ch only if additional constraints on v are met: $ch?v$ **where** $v \leq 3$.

I2 Mixing conditions and actions: a guard can invalidate actions already taken, e. g. for $tmp := f(x); guard(tmp);$ (despite their interdependency) the assignment to tmp only happens, if $guard$ is actually satisfied.

I3 A rich language of actions. The granularity of semantics is adjustable, allowing internal transitions and conditions on them, as well as collating effects of action sequences: **for** i **in** $0..1$ **do** $v[i] := 0;$ can be a *single* step, and semantically equivalent to $v[0] := 0;\ v[1] := 0$ without introducing observable intermediate steps.

I4 Avoidance of duplication of conditions: every conditional expression E_i is evaluated only once for **if** E_1 **then** C_1 **elsif** E_2 **then** C_2 **else** C_3 **end**, as opposed to a translation into transitions $s \xrightarrow{E_1/C_1} s'$, $s \xrightarrow{\mathbf{not}(E_1) \wedge E_2/C_2} s''$, $s \xrightarrow{\mathbf{not}(E_1) \wedge \mathbf{not}(E_2)/C_3} s'''$

We agree that these items are worthwhile requirements for an intermediate language, yet we would like to add the following points of practical importance:

I5 Tiny formal model for implementation, which is useful to assure correct implementation of state space generators.

I6 Extensionality, that helps to cope with additions to the formalism, like probabilistic aspects, timers, etc.

In addition to the above requirements, Garavel et al.[43] propose the intermediate language *New Technology Intermediate Format* (NTIF), which adheres to items I1–I4, and I6. However, NTIF has a natural rewriting based semantics. Similar as discussed in the previous subsection for rewrite engines, its implementation is likely to lack efficiency or requires sophisticated optimizations. Thus, NTIF does either not adhere to T2 (efficiency) or to I5.

Before we present and evaluate our proposed approach in light of these points in Sections 7.6.2 and 7.5, we review typical state space generation schemes in *parallel* verification tools like UppDMC (Section 4.1) or DIVSPIN [69].

6.3. Parallel State Space Generation

In the realms of parallelized verification, state space generators are constrained by additional requirements. To avoid bottlenecks, state space generation usually is carried out in parallel on multiple processors, often alongside the actual verification work, like in the *on-the-fly* algorithms we developed in Section 3.4.3. Processors circumvent work duplication by informing each other which parts have already been generated.

In commonly used approaches, processors save states assigned to them and distribute the remaining ones to their neighbors, which in turn proceed in the same way. This approach was proposed for parallel reachability analysis first by Nicol and Ciardo [78], and independently by Stern and Dill [90].

It has been reported often that this scheme scales reasonably well, and it is very natural and effortless to implement. However, state distribution implies that it must be possible to move states from one processor to another and resume generation of its successors at the new location. Thus, state representations should have low space profile, and must be self-sufficient in the sense that they contain all information needed for further processing. A conversion of location-independant binary representation is needed for storage and distribution, and this so-called *serialisation* should have low overhead.

Ideally, states should also support fast equality checking because this is a common operation in state space generation. If states can be treated as opaque objects, a good separation (from a software-engineering point of view) between state space generators and verification algorithms can be obtained, facilitating component reuse among tools.

In the remainder of this section, we briefly highlight how some existing tools for parallel state space generation work. The reader may note that none of them uses an intermediate format. Although we do not claim to be exhaustive here, we believe that most of currently existing tools fall in one of these classes.

PSPIN and PV. PSPIN [63] is a parallelized version of SPIN [48] which supports reachability analysis of state spaces described by the PROMELA modeling language. The authors reuse SPIN's PROMELA interpreter (or rather *interpreter compiler*) for state space generation. Through code inspection we found that for state serialisation they employ packing functions which capture the interpreters' current state in a memory buffer. Its contents are sent over a network to other processors and then unpacked again. Besides the disadvantagous serialisation overhead, we conjecture that this approach would carry over poorly into a multi-threaded environment, as it relies on global data structures, which need to be protected from concurrent mutations. A similar approach is used in the PV parallel verifier [80]. PV interprets an "extended subset of the PROMELA language" and is only able to check safety properties for this model.

Parallel Truth. As an experimentation platform for new concepts in parallel model checking research, we developed PARALLELTRUTH, a distributed prototype of the model checking tool TRUTH [15]. With PARALLELTRUTH, we are able to check properties given as alternation-free μ-calculus formulas *on-the-fly*. States are represented in an ad-hoc manner as CCS terms, and we reported that (de)serialisation efforts were mostly responsible for the immense overhead compared to a sequential variant. Besides that, the term rewriting used to generate successor states suffered the same run-time penalties that we discussed in Section 6.1.

UppDMC. An efficient implementation of our algorithms was presented by Holmén et al. [52] and is summarized in Section 4.1. State spaces are computed off-line by the μCRL toolkit. This allows for a *very* efficient encoding of states (basically enumerating them), but comes at a price: precalculation and storage often takes more time than actually carrying out the verification task. This approach also did not take advantage of the on-the-fly properties of our algorithms.

Next, we will present our virtual machine-based approach which not only overcomes most sources of overhead reported so far, but at the same time satisfies all the requirements detailed so far.

7. A Virtual Machine-based Approach

In this chapter, we carry out the design of a concrete virtual machine. We show that it adheres to the proposed design principles for intermediate representations. Furthermore, we show with experiments that it fulfils our expectations regarding its speed and usability in practice.

7.1. Virtual Machine Specification

Our virtual machine (VM) has a couple of features not all of which are commonly found at byte-code level in conventional VM architectures like the Java Virtual Machine (JVM) [84]. They are a superset of the features we observed as common in modeling languages in Section 6, in particular, we have:

Non-determinism If non-deterministic choice is encountered during executing, the machine offers all possible continuations to the scheduler who then decides which path to take.

Concurrency A built-in `run` byte-code allows to spawn processes at run-time.

Communication Both, rendezvous and asynchronous channel objects are provided for inter-process communication.

First-class channels Like in PROMELA and π-calculus [76], our machine allows channels to be sent over channels.

Priority scheme Our byte-code allows to specify which actions have to be given preference. Together with explicit control over externally visible actions, this allows to encode high-level constructs like PROMELA's `atomic` and `d_step`.

Speculative execution Certain code sequences are executed speculatively, and changes to the global state are rolled back if the sequence does not run to completion.

External Scheduling Scheduling decisions are delegated to host applications. This allows for implementation of different scheduling policies which is needed to cater for simulation (interactive scheduling) vs. state space exploration with some search strategy (breadth-first, depth-first, random, or combinations thereof).

The design of our VM was mainly driven by pragmatic decisions: it was our intention to create a model that is simple, efficient and embeddable as component into host applications, with implementation effort split between the VM and compilers targeting it. For example, many instructions make use of the VM's stack because it is trivial for compilers to generate stack-based code for expression evaluation. On the other hand, a stack-based architecture alone is inconvenient for translation of counting loops, thus registers were added. The RISC-like instruction set is motivated by the need for fast decoding inside the instruction dispatcher, the VM's most often executed routine.

Although our machine is a mixture of register-based and stack-based architecture, we are nevertheless dealing with finite state models in this paper, with concurrency modeled by interleaving semantics.

A complete specification of a virtual machine suitable as target for PROMELA is available [89]. Readers familiar with PROMELA will recognize its influence on some design decisions, making it easier to translate it into our byte-code language. However, in the interest of reusability we tried to keep these parts as generic as possible.

In the following we will present the virtual machine in detail. We start by specifying global and local state, and invariants which translations must preserve. Afterwards we present the byte-code semantics and how scheduling between alternatives is done.

7.1.1. Machine State

The machine's global state as depicted in Figure 7.1 consists of a few global objects and the local state of its processes.

Definition 7.1.1 (Global State)
The *global state* $\Gamma = (\Pi, e, G, \Phi)$ of our virtual machine is a tuple

$$\Gamma \in \mathit{Processes} \times \mathit{Pid}_{\perp} \times \mathit{Mem} \times \mathit{Channels}$$

with Π denoting a finite set of processes, e the process identifier of a process with exclusive execution privileges ($\perp$ if none), G the global variable store, and Φ the—again finite—set of existing channels (channels are global objects).

We will refer to the set of all global states as Γ as well, if the context makes clear what is meant.

Definition 7.1.2 (Process)
A *process* $\pi = (p, M, \Lambda')$ is a tuple

$$\pi \in \mathit{Processes} = (\mathit{Pid} \times \mathit{ExecMode} \times \mathit{ProcessState}') \cup \{\mathrm{stop}\}$$

with p denoting a globally unique identifier, $M \in \{\underline{\mathrm{N}}, \underline{\mathrm{A}}, \underline{\mathrm{I}}, \underline{\mathrm{T}}\}$ its execution mode (normal, atomic, invisible, terminated), and Λ' the local state of a process (Definition 7.1.4).

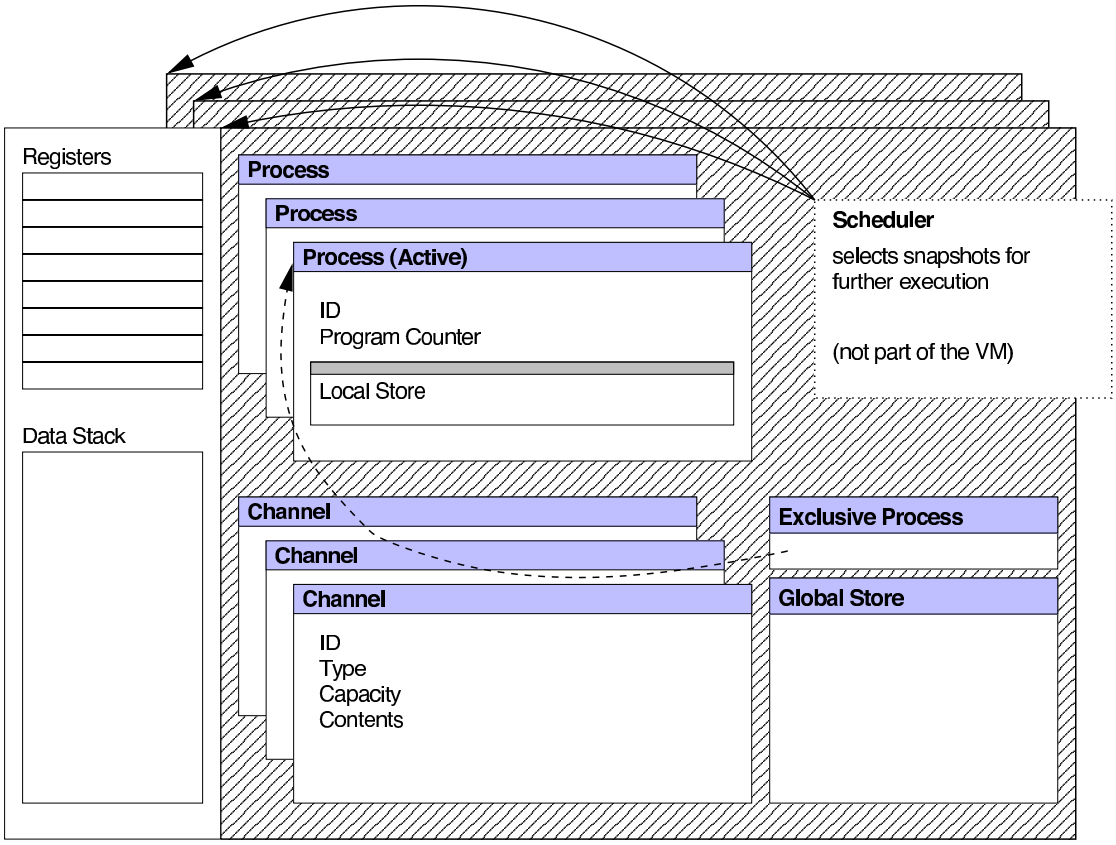


Figure 7.1.: Overview over the state of the virtual machine. Note that there is only one set of registers and one stack, as only a single process can be active at any given point. The scheduler is provided by external sources (for example, a model checker). It restores a snapshot of the VM's state and resumes execution from there.

Furthermore, we allow the special symbol stop to denote a *deadlocked* process which cannot make any further step.

Note that while a single process can be deadlocked, there might be others which can still continue, so that there is no *global deadlock* yet.

Remark 7.1.3
Often, we do not want a global state $\Gamma = (\Pi, e, G, \Phi)$ to contain the deadlocked process stop. To simplify notation, we write $\Gamma \neq \mathrm{stop}$ iff no process in Π is deadlocked: $\forall \pi \in \Pi : \ \pi \neq \mathrm{stop}$.

A process is either *inactive* or *active*. In the latter case Λ' is augmented as shown below.

Definition 7.1.4 (Local Process State)
A *local process state* $\Lambda' = (L, m)$ is a pair

$$\Lambda' \in \mathit{ProcessState}' = \mathit{Mem} \times \mathbb{N}$$

and denotes the process-local variable store L and its program counter m.

When a process becomes active, its state Λ' is augmented with registers R_0 and a stack $D_\epsilon = \epsilon$ to its *active local state* $\Lambda = (L, m, R_0, D_\epsilon)$:

$$\Lambda \in \mathit{ProcessState} = \mathit{Mem} \times \mathbb{N} \times \mathit{Registers} \times \mathit{Stack}$$

When it becomes inactive again, its last two components are projected away.

Definition 7.1.5 (Store)
We identify three *stores* in our virtual machine model: for global (G) and local variables (L), and for registers (R). As usual, we model stores as mappings $\sigma \in \mathbb{N} \to \mathit{Value}$, that is for a store σ, $\sigma[i]$ denotes the store's value at position i. Replacing a value v at position i in the store is written as $\sigma[i/v]$ and yields a new store σ':

$$\sigma' := \sigma[i/v] \text{ and } \sigma'[j] = \begin{cases} v & \text{if } i = j \\ \sigma[j] & \text{otherwise} \end{cases}$$

Initial stores are denoted as σ_0 ($\forall i : \ \sigma_0[i] := 0$). For convenience, we write r_i to reference the ith register $R[i]$.

We added registers to our virtual machine for situations when byte-code effects on the machine's state are not fitting well to a stack model, for instance if values are operated on more than once.

Definition 7.1.6 (Data Stack)
Expression evaluation takes place on the *data stack* component $D \in \mathit{Stack} = \mathit{Value}^*$ of a process state. A stack is represented as finite (possibly empty) word $D = v_n : \cdots : v_1, \quad v_i \in \mathit{Value}, n \in \mathbb{N}$.

We denote the empty stack as $D_\epsilon = \epsilon$.

Communication

Processes can use several ways to communicate values among each other. First, they can use the global store G which can be modified by any process at any time. A more structured way of communication is provided by means of *channels*. They also offer a model for message-passing synchronization. In our machine, communication channels are typed and bounded, and we distinguish between rendezvous channels and asynchronous channels.

Definition 7.1.7
A *channel* $\varphi = (c, l, t, C)$ is a tuple

$$\varphi \in \mathit{Channels} = \mathit{ChanId} \times \mathbb{N} \times \mathbb{N} \times \mathit{Message}^*$$

with c denoting a globally unique channel identifier, l the channel capacity, and $C = c_0 : \cdots : c_l$ its current contents (c_l being the last message in the channel). Each message $c_i \in \mathit{Message} = \mathit{Value}^*$ consists of a sequence of values of length t.

Rendezvous channels have zero capacity. A message can temporarily be stored in a channel during rendezvous communication, hence exceeding the capacity of the channel. Such states are internal to the virtual machine and unobservable to its outside. Similarly, asynchronous channels which exceed their capacity automatically fall back to the same behavior as rendezvous channels: send operations on those block until they are within their allowed capacity again.

Definition 7.1.8 (Rendezvous Communication)
We define a predicate $\mathrm{synch}(\Gamma)$ on a global state $\Gamma = (\Pi, e, G, \Phi)$ to determine whether rendezvous communication is taking place: at least one channel $\varphi = (c, l, t, C)$ contains more messages than its capacity l allows.

$$\mathrm{sync}(\Gamma) := \begin{cases} \mathrm{false} & \text{if } \forall \varphi = (c, l, t, C) \in \Phi : \ |C| \leq l \\ \mathrm{true} & \text{otherwise} \end{cases}$$

7.1.2. Invariants

Translation to our byte-code language must guarantee the following invariants: as already pointed out in Definition 7.1.4, a process becoming active again always resumes execution with register set R_0 and the empty stack D_ϵ. Conversely, at those points in the program when a process may become inactive, the contents of registers and stack are discarded and need not matter for the rest of its execution.

Because the number of local variables is fixed, a local state Λ' hence occupies constant space only.

7.1.3. Byte-code Semantics

Having defined the state of our virtual machine, we now proceed by defining the semantics of operations on it. These operations are carried out at process level, with only a single process being active at once.

In the spirit of an earlier attempt to define the semantics of PROMELA by Holzmann and Natarajan [55], we compose our semantics from several smaller parts by defining five relations to model process activation, internal and prioritized transitions, intermediate and finally scheduler transitions.

A transition from state Γ_1 to Γ_2 is a relation $\rightarrow_T \in \Gamma \times \Sigma_T \times \Gamma$, with a finite set of labels Σ_T and set of states Γ. If not important, we will elide labels from our presentation. For brevity, we generally write $\Lambda_1, G_1, \Phi_1 \rightarrow \Lambda_2, G_2, \Phi_2$ instead of

$$\begin{aligned}(\{(p, M, \Lambda_1), \pi_1, \ldots, \pi_n\}, e, G_1, \Phi_1) \\ \rightarrow (\{(p, M, \Lambda_2), \pi_1, \ldots, \pi_n\}, e, G_2, \Phi_2) \\ \pi_i = (p_i, M_i, (L_i, m_i)) \text{ for all } 1 \leq i \leq n\end{aligned}$$

State components remaining unchanged in a transition are left out.

As mentioned before, only one process can be active at any point in time. Thus we define process activation as transition

$$\begin{aligned}(\{(p, M, (L, m)), \pi_1, \ldots, \pi_n\}, e, G, \Phi) \\ \xrightarrow{p}_{act} (\{(p, M, (L, m, R_0, D_\epsilon)), \pi_1, \ldots, \pi_n\}, e, G, \Phi) \\ \forall i \in \{1, \ldots, n\} : \pi_i = (p_i, M_i, (L_i, m_i)) \\ \text{and } e \in \{p, \perp\}, M \neq \underline{\mathrm{T}}\end{aligned}$$

A process needing exclusive execution privileges *must* be activated, otherwise any process can be activated ($e = \perp$). Processes already run to completion ($M = \underline{\mathrm{T}}$) are not activated again.

Next, we define those transitions an active process can possibly take: the *internal-step* relation $\rightarrow_{int} \in \Gamma \times \Gamma$ is the least relation satisfying the rules given in the following. For reasons of presentation, we divided internal steps into several categories. Note that the byte-code operation to be executed next is determined by indexing program counter m of the currently active process into a global instruction list $\underline{\mathrm{Instr}}$.

Load and Store

Our machine supports usual operations to load constants (`LDC`), and manipulate values of local and global variables (`LDV`, `STV`), as defined in Table 7.1. To avoid stack juggling

LDC c	load constant c onto top of data stack $(L, m, R, D) \rightarrow_{int} (L, m+1, R, D : c)$
LDV g	load variable onto top of data stack $(L, m, R, D : a) \rightarrow_{int} (L, m+1, R, D : L[a])$ if $g = \underline{\mathrm{L}}$ $(L, m, R, D : a), G \rightarrow_{int} (L, m+1, R, D : G[a]), G$ if $g = \underline{\mathrm{G}}$
STV g	store stack top in variable $(L, m, R, D : v : a) \rightarrow_{int} (L[a/v], m+1, R, D)$ if $g = \underline{\mathrm{L}}$ $(L, m, R, D : v : a), G \rightarrow_{int} (L, m+1, R, D), G[a/v]$ if $g = \underline{\mathrm{G}}$
POP r_i	pop top-most value from stack into register $(L, m, R, D : v) \rightarrow_{int} (L, m+1, R[i/v], D)$
PUSH r_i	push value from register onto stack $(L, m, R, D) \rightarrow_{int} (L, m+1, R, D : r_i)$

Table 7.1.: Load and Store byte-codes

operations like DUP, SWAP, etc., values can be stored into and retrieved from registers with PUSH and POP.

Arithmetic and Boolean Operations

Expression byte-codes like ADD, LT, AND, NEG etc. operate on one or more of the stack's top-most entries. Their semantics are obvious and thus only defined exemplarily:

$$\mathtt{OP}_{\otimes}: \quad (L, m, R, D : u : v) \rightarrow_{int} (L, m+1, R, D : u \otimes v)$$

Control-flow Operations

For control flow changes, we define conditional and unconditional jumps in Table 7.2. In order to allow explicit modeling of non-determinism, we define NDET a as having two possible successor states: one continuing with the next instruction and the other continuing at instruction a. In some situations, it is helpful to allow *conditional non-determinism*, where the existence of one alternative is dependent on the presence or absence of another. For this, we add byte-codes ELSE a and its dual UNLESS a.

Operations on Channels

In Table 7.3 we introduce several operations on communication channels. These include operations to dynamically create channels, query their properties, and manipulate their contents. All of them require a channel identifier on the stack. They operate on both types of channels, with rendezvous channels special-cased in the definition of CHADD

JMP a	unconditional jump $(L, m, R, D) \rightarrow_{int} (L, a, R, D)$
JMPNZ a	jump if non-zero $(L, m, R, D : 0) \rightarrow_{int} (L, m+1, R, D)$ $(L, m, R, D : v) \rightarrow_{int} (L, a, R, D)$, if $v \neq 0$
NDET a	non-deterministic jump $(L, m, R, D) \rightarrow_{int} (L, m+1, R, D)$ $(L, m, R, D) \rightarrow_{int} (L, a, R, D)$
ELSE a	else jump $(L, m, R, D) \rightarrow_{int} (L, m+1, R, D)$ $(L, m, R, D) \rightarrow_{int} (L, a, R, D)$ if $(L, m+1, R, D) \rightarrow^*_{int} \Lambda' \rightarrow_{end}$ stop
UNLESS a	unless jump $(L, m, R, D) \rightarrow_{int} (L, a, R, D)$ $(L, m, R, D) \rightarrow_{int} (L, m+1, R, D)$ if $(L, a, R, D) \rightarrow^*_{int} \Lambda' \rightarrow_{end}$ stop

Table 7.2.: Control-flow byte-codes

($k < \max(l, 1)$). In consequence, synchronous communication is done with an intermediate (but invisible) step. We will return to this topic in section 7.1.4.

Sending in *First-In First-Out* (FIFO) order is done by allocating a new message with CHADD and settings its contents with CHSET. Message reception is carried out in two steps as well: reading the contents of a message and then deleting it. The rationale here is that we are able to handle channel queries (full/emptiness) with the same byte-codes, and with the addition of four more byte-codes any of PROMELA's rather uncommon channel operations.

Spawning New Processes

To start a new process, its current parameters are placed onto the data stack. Specifying the size of these parameters and the start address of its code, a new process is instantiated:

RUN k, a run a new process starting at address a

$$(\{\pi, \pi_1, \ldots, \pi_n\}, e, G, \Phi) \rightarrow_{int} (\{\pi', \pi_1, \ldots, \pi_n, \pi''\}, e, G, \Phi)$$

with $\pi = (p, M, (L, m, R, D : v_0 : \cdots : v_{k-1}))$
and $\pi' = (p, M, (L, m+1, R, D : p''))$
and $\pi'' = (p'', \underline{\mathrm{N}}, (L_0[0/v_0, \ldots, k-1/v_{k-1}], a))$
and $p'' \in Pid$ a unique process identifier

`CHNEW` l, t	create a new empty channel with maximum length l and message size t $(L, m, R, D), \Phi \rightarrow_{int} (L, m+1, R, D : c), \Phi \cup \{(c, l, t, \epsilon)\}$ with c a unique channel identifier
`CHMAX`	get maximum channel length $(L, m, R, D : c), \Phi \rightarrow_{int} (L, m+1, R, D : \max(1, l)), \Phi$ if $(c, l, t, C) \in \Phi$
`CHLEN`	get channel length (number of messages in channel) $(L, m, R, D : c), \Phi \rightarrow_{int} (L, m+1, R, D : k), \Phi$ if $(c, l, t, c_1 : \cdots : c_k) \in \Phi$
`CHADD`	allocate new message in channel $(L, m, R, D : c), \Phi \rightarrow_{int} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ if $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, c_1 : \cdots : c_{k+1})$, $\varphi \in \Phi$ and $k < \max(1, l)$, $c_{k+1} = 0^t$
`CHSET`	set values in last channel message $(L, m, R, D : c : o : v), \Phi \rightarrow_{int} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ if $\varphi = (c, l, t, C : c_k)$, $\varphi' = (c, l, t, C : c'_k)$, $\varphi \in \Phi$ and $c_k = (v_0, \ldots, v_{t-1})$, $c'_k = (v'_0, \ldots, v'_{t-1})$ $v'_o = v$ if $o < t$, $v'_i = v_i$ for $i \neq o$
`CHGET`	get value from first channel message $(L, m, R, D : c : o), \Phi \rightarrow_{int} (L, m+1, R, D : v), \Phi$ if $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi \in \Phi$ and $k > 0$, $c_1 = (v_0, \ldots, v_{t-1})$, $v = v_o$ if $o < t$, $v = 0$ otherwise
`CHDEL`	delete first message from channel $(L, m, R, D : c), \Phi \rightarrow_{int} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ if $\varphi = (c, l, t, c_1 : C) \in \Phi$, $\varphi' = (c, l, t, C)$
`CHSORT`	sort last message into channel $(L, m, R, D : c), \Phi \rightarrow_{int} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ if $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, c_1 : \cdots : c_{j-1} : c_k : c_j : \cdots : c_{k-1})$, $\varphi \in \Phi$ and $k > 0$, $j = \min(\{j' \mid 1 \leq j' < k, c_{j'} > c_k\} \cup \{k\})$
`CHROT`	rotate messages in channel $(L, m, R, D : c), \Phi \rightarrow_{int} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ if $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, : c_2 : \cdots : c_k : c_1)$, $\varphi \in \Phi$

Table 7.3.: Operations on channels

`STEP` M'	step complete with mode M' $(\{(p, M, (L, m, R, D))\} \cup \Pi, e, G, \Phi)$ $\xrightarrow{M'}_{end} (\{(p, M', (L, m+1))\} \cup \Pi, e', G, \Phi)$ $e' := \begin{cases} p & \text{if } M' \in \{\underline{\mathrm{A}}, \underline{\mathrm{I}}\} \\ \bot & \text{otherwise} \end{cases}$ and $\forall \pi_i \in \Pi : \ \pi_i = (p_i, M_i, (L_i, m_i))$
`NEX`	step not executable $(L, m, R, D) \not\rightarrow_{end} \mathrm{stop}$

Table 7.4.: Operations for Process Deactivation

Deactivation of Processes

Following a *cooperative multitasking* approach, eventually a process allows resumption of other processes by deactivating itself with one of the operations in Table 7.4.

We introduce `STEP` M' as flexible means to control which states become visible to an external scheduler. If further execution of a process is not anticipated (e.g. because of unsatisfied guard conditions or reception attempts on empty channels), process execution may be aborted explicitly by `NEX`. This byte-code instruction can be used to translate *guards*—boolean conditions which can enable or disable a transition.

7.1.4. Scheduling

With all the machinery in place, we now proceed with the relation of *scheduler transitions*, $\rightarrow_{sched}$. We define it in terms of *intermediate transitions* $\rightarrow_{step}$, which is the least relation satisfying

$$\Gamma \xrightarrow{p,M}_{step} \Gamma' \quad \text{if} \quad \Gamma \xrightarrow{p}_{act} \Gamma_0 \rightarrow^*_{int} \Gamma_1 \xrightarrow{M}_{end} \Gamma'$$

This means, that in a machine state Γ some process identified as p is activated, then a number of internal transitions happen, until at some point the process deactivates itself in state Γ', giving the whole sequence mode M.

In case the machine gets "stuck" without successor states because some process with exclusive execution privileges becomes deadlocked, this process loses them, thus enabling execution possibilities for other processes:

$$(\Pi, e, G, \Phi) \xrightarrow{p,M}_{step} \Gamma' \quad \text{if} \quad (\Pi, e, G, \Phi) \xrightarrow{e,-}_{step} \mathrm{stop} \quad \text{and} \quad (\Pi, \bot, G, \Phi) \xrightarrow{p,M}_{step} \Gamma'$$

We can then define the transitions visible to an external scheduler. The approach we took is due to our decision to model rendezvous communication within the interleaving model and thus using an intermediate state which is not revealed to the scheduler. We can distinguish three cases: a process ends a sequence of invisible steps with either a visible transition or a transition leading to deadlock, and no interim rendezvous communication can take place, or, rendezvous communication can take place, with the restriction that the sending and receiving halves of the communication must be consecutive.

Definition 7.1.9 (Scheduler Transition)
We define the *scheduler transition* relation $\xrightarrow{p}_{sched}$ as least relation satisfying the following rules.

- A scheduler transition consists of a (possibly empty) sequence of invisible steps, followed by a visible step, that is, a step with mode $\underline{\mathrm{N}}$ (normal), $\underline{\mathrm{A}}$ (atomic) or $\underline{\mathrm{T}}$ (terminated). None of the steps is a rendezvous communication.

$$\Gamma \xrightarrow{p}_{sched} \Gamma' \text{ if } \Gamma = \Gamma_1 \xrightarrow{p,\underline{\mathrm{I}}}_{step} \cdots \xrightarrow{p,\underline{\mathrm{I}}}_{step} \Gamma_{n-1} \xrightarrow{p,M}_{step} \Gamma_n = \Gamma' \\ \text{and } \forall i: \ \neg\mathrm{sync}(\Gamma_i) \text{ and } M \neq \underline{\mathrm{I}} \text{ and } \Gamma' \neq \mathrm{stop}$$

- Alternatively, if a sequence of invisible steps leads to a deadlocked process, the last step right before the deadlock becomes visible *irrespectively of its mode* $\underline{\mathrm{I}}$.

$$\Gamma \xrightarrow{p}_{sched} \Gamma' \text{ if } \Gamma = \Gamma_1 \xrightarrow{p,\underline{\mathrm{I}}}_{step} \cdots \xrightarrow{p,\underline{\mathrm{I}}}_{step} \Gamma_{n-1} \xrightarrow{p,\text{-}}_{step} \mathrm{stop} \\ \text{and } \forall i: \ \neg\mathrm{sync}(\Gamma_i) \text{ and } \Gamma' = \Gamma_{n-1}$$

- Lastly, we allow a rendezvous channel to actually contain one message more than its capacity allows, if the immediately following transition resolves this again by having a rendezvous partner (different from the sender) receiving this message, so that said rendezvous channel becomes empty again and the resulting state becomes visible to the scheduler again. In this case the sender loses its execution privilege. It can then be picked up by the receiver. Note that we do not allow a process to have rendezvous communication with itself ($p \neq p'$).

  With this mechanism, rendezvous communication can be used to pass around

execution privileges between processes.

$$\Gamma \xrightarrow{p}_{sched} \Gamma'' \text{ if } \Gamma \xrightarrow{p,M}_{step} \Gamma' = (\Pi', e', G', \Phi') \text{ and } (\Pi', \bot, G', \Phi') \xrightarrow{p',M'}_{step} \Gamma'' \text{ and } \mathrm{sync}(\Gamma') \text{ and } \neg\mathrm{sync}(\Gamma'') \text{ and } p \neq p' \text{ and } \Gamma'' \neq \mathrm{stop} \text{ and } M \neq \underline{\mathrm{T}}$$

In all cases, we do not allow a scheduler transition to lead to a global state containing a deadlock process stop.

Our handling of deadlock processes allows us to define a global deadlock state Γ where no process can complete a scheduler transition naturally: there is no Γ' such that $\Gamma \xrightarrow{p}_{sched} \Gamma'$.

Definition 7.1.10 (Initial State)
The scheduler starts program execution with the initial state of our machine

$$\Gamma_{init} = (\{(1, \underline{\mathrm{N}}, (L_0, init))\}, \bot, G_0, \varnothing)$$

7.2. State Space Generation

State space generation can be carried out straight-forwardly with our virtual machine. Given a byte-code program and an initial state, a minimal interface of our virtual machine for building a model's state space is some function

$$\texttt{next-state} : State \to 2^{State}$$

which is naturally induced by the scheduler relation $\rightarrow_{sched}$.

7.3. Use Case: PROMELA

We validated our virtual machine-based approach to state space generation, by defining a translation from PROMELA to byte-code, thus defining its operational behaviour through the backdoor. A complete translation procedure is given by Schürmans [89]). Although other modeling languages could have been used just as well, PROMELA was chosen because it is a truly non-trivial example and it has wide acceptance inside and outside academia.

To give an impression of the translation into byte-code, a very short example that shows a few interesting aspects is presented in Figure 7.2. The corresponding byte-code is shown in Figure 7.3.

The PROMELA program specifies a sender process `send`, which sends values 3 and 5 through a channel `c` of capacity 1 integer. The respective byte-code for both sending operations starts at $send$ and se_2, they are separated by `STEP` A due to the `atomic` keyword in the PROMELA program which denotes that no concurrent action may take place between the two send operations.

The corresponding receiver process `recv` receives messages from channel `c` in a loop until the received integer value `i` is bigger than 5. When translated to byte-code, instruction `ELSE` re_4 denotes that if the following code executes an `NEX` before some `STEP` M' (in this case, the one at label re_2), execution is resumed at re_4. The byte-code generated for guard `i < 5` is between labels re_0 and re_2, the body follows up to label re_4. The remaining code is the landing pad for the jump resulting from the `else` guard, and for closing the loop. Both processes are started from label $init$.

Note, that in our byte-code, all potentially blocking actions, as well as all steps which result in scheduler transitions are explicit through `NEX` and `STEP` M', respectively.

When executed, the receiver blocks until a message is in the channel. The sender writes value 3 into the channel, and blocks while trying to write 5 into the channel, as it is full. The sender's blocking causes it to lose its atomic execution priviledges, and hence the receiver can continue. It is unblocked because there is a message in the channel available. The receiver then receives 3, resumes the loop and blocks again, as the channel is now empty again. This allows the sender to send 5 and terminate. The receiver unblocks, receives 5 and stores it into local variable `i`. In the next loop iteration, guard `i < 5` blocks, thus allowing the `else` guard to trigger, which subsequently break out of the loop, and the receiver terminates as well.

7.4. Benchmarks

We implemented the virtual machine[1] sketched in the previous section to confirm the practicality of our approach. Our efforts resulted in around 5,000 lines of commented C code, including usage examples, which is rather small for a virtual machine. It turns out that this prototype performs competitively even when compared to state-of-the-art tools like SPIN.

Contrary to SPIN, the sole task of our VM is state space generation. Additional functionality like model checking, possibly together with, for example, *partial order reduction* [24] is duty of other components not covered here.

[1] called NIPS VM, New Implementation of PROMELA semantics, due to its beginnings

```
chan c = [1] of {int};
active proctype send() {
  atomic {
    c!3;
    c!5
  }
}
```

```
active proctype recv() {
  int i = 2;
  do
  :: i < 5; c?i
  :: else; break
  od
}
```

Figure 7.2.: PROMELA code for a sender process `send`, which sends values 3 and 5 through a channel `c` of capacity 1 (integer). The corresponding receiver process `recv` receives messages from channel `c` in a loop until the received integer value `i` is bigger than 5.

```
                send : LDC 0          se2 : LDC 0          recv : LDC 2          JMPNZ re3
                       LDV G                 LDV G                 LDC 0          NEX
                       POP r0                POP r0                STV L    re3 : PUSH r0
                       PUSH r0               PUSH r0       re0 : ELSE re4        PUSH r0
                       CHLEN                 CHLEN                 LDC 0          LDC 0
init : CHNEW 1,1       PUSH r0               PUSH r0               LDV L          CHGET
       LDC 0           CHMAX                 CHMAX                 LDC 5          LDC 0
       STV G           LT                    LT                    LT             STV L
       RUN 0, send     JMPNZ se1             JMPNZ se3             JMPNZ re2      CHDEL
       POP r0          NEX                   NEX                   NEX            JMP re5
       RUN 0, recv  se1 : PUSH r0     se3 : PUSH r0       re2 : STEP N     re4 : STEPN
       POP r0          CHADD                 CHADD                 LDC 0          JMP re1
       STEPT           PUSH r0               PUSH r0               LDV G    re5 : STEPN
                       LDC 0                 LDC 0                 POP r0         JMP re0
                       LDC 3                 LDC 5                 PUSH r0  re1 : STEPT
                       CHSET                 CHSET                 CHLEN
                       STEPA                 STEPT
```

Figure 7.3.: Byte-code translation for the PROMELA program in Figure 7.2. Entry points are label *init* for the program, label *send* and *recv* for the respective processes, which are both started from the *init* process.

For our comparison to SPIN we employ standard breadth-first search with full state space storage. We used the same hash function as SPIN does (due to Jenkins [58]).

While correctness, ease of reasoning and implementability are already worthwhile traits of our virtual machine, it is also important that a state space generator is fast in practice. In order to get a meaningful idea on the speed of our VM, we conducted extensive experiments and measured the rate with which successor states are generated (Table 7.5).

Unfortunately, we cannot compare state space size (and thus run-time) directly, because in general we generate slightly more states than SPIN due to our finer-grained program counter. We expect this to be cleaned up by subsequent optimizations passes in our compiler by reducing the number of visible STEP M instructions ($M \in \{\underline{\mathrm{N}}, \underline{\mathrm{A}}\}$). One such optimization, called *path compression* [98], has been extended to deal with additional features provided by our virtual machine model: dynamic process creation, asynchronous communication channels, control-flow non-determinism and speculative execution. A preliminary implementation shows promising results (Table 7.5). Besides this, we would like to highlight our NIPS implementation itself is not optimized at all, in contrast to SPIN which has been under steady development for more than 15 years, by the time of writing.

Our test setup consisted of an AMD Athlon 64 3500+ running Linux. We used SPIN 4.2.5 for comparison. SPIN translates PROMELA models into C source code which subsequently is compiled, and then run for the analysis.

By default, SPIN uses data-flow optimizations and *statement merging* [53] to reduce size of the *explored state space*, thus requiring less time and memory for the task. The optimizations can be disabled optionally (`spin -o1 -o3`).

We benchmarked SPIN without said optimizations against our virtual machine implementation (columns "Unoptimized" in Table 7.5), and another time with both optimizations enabled, against our unmodified virtual machine, but with path compression enabled in our PROMELA compiler.

We compiled the `pan.c` files generated by SPIN from the PROMELA models, and used gcc (version 3.3.5) with option `-O2` (C optimisations), `-DNOREDUCE` (disabling partial-order reduction) and `-DBFS` (enabling breadth-first search). The resulting executable was used for benchmarking. Note, that our VM interprets instructions while `pan.c` is *compiled* into a native executable.

In our tests we used models that come with the SPIN distribution. Our experiments show that NIPS (version 1.2.2) is competitive to SPIN both in state size (rightmost columns of Table 7.5) and state space generation speed.

The size of states, which contain all information needed to restart the virtual machine from (global and local variables, channels, processes), is typically within a few bytes of what SPIN reports as state size.

Comparisons of state space size and run-times require further explanations. For very

Parameter		NIPS Virtual Machine: Unoptimized States	Unoptimized Time	Unoptimized States/sec.	NIPS Virtual Machine: with Path Compression States	with Path Compression Time	with Path Compression States/sec.	SPIN: Unoptimized States	Unoptimized Time	Unoptimized States/sec.	SPIN: Data-Flow Opt., Statement Merging States	Data-Flow Opt., Statement Merging Time	Data-Flow Opt., Statement Merging States/sec.	NIPS State size in bytes	SPIN State size in bytes
MAX							**eratosthenes**								
	6	170	0.002	76853.53	34	0.001	51593.32	195	0.016	12187.50	128	0.016	8000.00	130	124
	10	764	0.020	38055.39	74	0.003	26047.17	1006	0.018	55888.89	548	0.018	30444.44	163	156
	14	2744	0.051	53339.55	190	0.006	33009.03	3864	0.026	148615.38	2263	0.026	87038.46	229	220
	18	7766	0.166	46893.02	342	0.012	28023.60	12035	0.058	207500.00	6477	0.058	111672.41	262	252
	22	24092	0.569	42364.48	626	0.025	24584.69	41610	0.344	120959.30	21539	0.344	62613.37	295	284
	26	69920	1.717	40730.78	1162	0.054	21361.08	129823	2.430	53425.10	69618	0.430	161902.33	328	316
	30	146222	3.824	38237.95	1710	0.088	19514.09	282914	11.855	23864.53	130062	3.855	33738.52	361	348
	34	347012	10.418	33307.78	2914	0.177	16451.75	713817	171.441	4163.63	342028	26.441	12935.52	394	380
N	L						**leader**								
3	6	754	0.009	79410.22	105	0.002	59965.73	743	0.018	41277.78	407	0.018	22611.11	131	116
4	8	5678	0.082	69216.89	379	0.008	47889.82	5626	0.037	152054.05	2410	0.037	65135.14	186	180
5	10	46091	0.649	70986.55	1509	0.035	43504.58	45937	0.268	171406.72	15791	0.268	58921.64	249	220
6	12	382465	6.180	61891.22	6241	0.176	35533.72	382151	3.120	122484.29	106449	0.120	887075.00	320	308
N	L						**leader2**								
3	6	4571	0.054	85268.71	667	0.010	63907.25	4476	0.027	165777.78	2430	0.027	90000.00	138	124
4	8	143373	1.321	108507.81	10012	0.161	62102.25	142260	0.650	218861.54	60052	0.650	92387.69	193	188
N							**peterson_N**								
	2	327	0.003	109879.03	30	0.000	72289.16	303	0.017	17823.53	185	0.017	10882.35	38	40
	3	51118	0.268	190879.12	853	0.012	71112.96	45927	0.085	540317.65	25371	0.085	298482.35	50	48
							pftp								
		1378184	10.033	137368.71	301603	4.996	60372.40	1275180	3.770	338244.03	219167	0.770	284632.47	189	152
							snoopy								
		124434	2.385	52180.87	68658	1.442	47603.20	91925	0.436	210837.16	61624	0.436	141339.45	205	188
N							**sort**								
	5	21245	0.276	76930.88	572	0.010	54533.32	14349	0.077	186350.65	4652	0.077	60415.58	181	184
	6	152628	1.789	85331.92	2019	0.040	49967.83	95677	0.576	166105.90	22350	0.576	38802.08	215	216

Table 7.5.: State Space Generation: A comparison between NIPS and SPIN. PROMELA models are taken from the SPIN distribution. Times are measured as wall-clock time in seconds on an AMD Athlon 64 3500+ running Linux.

```
$ nips_vm -Rq eratosthenes.pr.b

NIPS VM - New Implementation of Promela Semantics Virtual Machine
version 1.2.2 date 2005-09-30
Copyright (C) 2005: Stefan Schuermans <stefan@schuermans.info>
                    Michael Weber <michaelw@i2.informatik.rwth-aachen.de>
                    Lehrstuhl fuer Informatik II, RWTH Aachen
Copyleft: GNU public license - http://www.gnu.org/copyleft/gpl.html

MSC: 2 is prime
MSC: 3 is prime
MSC: 4 = 2*2
MSC: 6 = 2*3
MSC: 5 is prime
MSC: 8 = 2*4
MSC: 7 is prime
MSC: 9 = 3*3
MSC: 10 = 2*5
MSC: 11 is prime
MSC: 12 = 2*6
MSC: 14 = 2*7
MSC: 13 is prime
MSC: 15 = 3*5
MSC: 16 = 2*8
MSC: 18 = 2*9
MSC: 17 is prime
MSC: 20 = 2*10
MSC: 19 is prime
MSC: 21 = 3*7
MSC: 22 = 2*11
MSC: 24 = 2*12
MSC: 26 = 2*13
MSC: 25 = 5*5
MSC: 23 is prime
```

Figure 7.4.: Example output for a random walk through the state space of `eratosthenes(26)` with NIPS. The order of lines can differ in several runs due to concurrency of the sieve processes.

small state spaces (below a few thousand states), differences in run-times are negligible, as results are almost instantaneous, and dominated by load times, among other things. On larger models SPIN seems to be around a factor of 2–3 times as fast as our VM with optimizations disable. Notable exceptions are instances of the `eratosthenes` model for big values of parameter `MAX`, for which NIPS outperforms SPIN in terms of run-time by more than factor 16. The difference in state space size seems to be due to the peculiar way how SPIN counts states. We have not been able to gain further insight on the reasoning behind it.

Note that despite the differences in state count we do have a convincing argument that the `eratosthenes` model is still doing what it is supposed to—calculating prime numbers up to a given limit. As they are printed out, we can assure ourselves that indeed

no vital functionality is thrown away and the explored state space differs from the actual state space only in indistinguishable elisions of interleavings [98].

When comparing both tools with optimizations turned on, NIPS wins in terms of state space size on many of the tested models, sometimes generating only fractions of the state space when compared to the results of SPIN with optimizations turned on (0.9% for `eratosthenes(34)`, 5.9% for `leader(6,12)`, 3.4% for `peterson_N(3)`), resulting in reduced run-times as well.

Exceptions in which SPIN still wins over NIPS with respect to state space size and run-time are the `snoopy` and `pftp` models. Although the path compression optimization manages to reduce the explored state space to less than 75%, there is still room left for improvements by additional optimizations which we did not pursue yet.

Overall, speed of state space generation seems already good enough for our purposes for now, especially if taken into account that we are benchmarking interpreted execution of our byte-code language against compiled code of SPIN's `pan.c` generator. For example, state space generation costs with our VM is already insignificant compared to the communication costs in our distributed model checker.

7.5. Evaluation as Intermediate Language

We will now review our virtual machine design with regard to the requirements proposed in Chapter 6.

One of the design goals for our byte-code language was that it must be easy and fast for a machine to execute. We have experimentally shown in Section 7.4 that we are able to reach this goal indeed.

Translation from our use case example PROMELA is automatic (requirement T1), as one would expect.

As our compiler does not generate the state space of an input program itself, but instead creates a byte-code program which when executed creates the state space, we can claim time and space efficiency (T2). Each PROMELA construct can be expressed in a short sequence of byte-code operations, and due to its binary format, the byte-code's size is larger than the input program's source code by only a small constant factor, usually less than 2. As far as we can tell, our approach is expressive enough to deal with features commonly found in specification languages, certainly it is for PROMELA (T3).

Finally, the last requirement on the translation process is the possibility to present results in terms of the input formalism, as this is what users would expect (T4). We achieve this with standard compiler construction and debugging techniques, and preserve enough information throughout the compilation process to relate each byte-code operation back to the high-level construct that caused the compiler to emit it.

We can then turn our attention to requirements on the intermediate language itself. Our byte-code language allows for arbitrary guards to be executed, thus we are able to retroactively invalidate statements. The example $ch?v$ **where** $v \leq 3$ from requirement I1 could for example be translated by using a suitable statement translation function $\mathcal{C}[\![\cdot]\!]$ and an expression translation function $\mathcal{E}[\![\cdot]\!]$:

$$\begin{aligned} \mathcal{C}[\![ch?v \textbf{ where } v \leq 3]\!] := \quad & \mathcal{C}[\![ch?v]\!] \\ & \mathcal{E}[\![v \leq 3]\!] \\ & \texttt{JMPNZ } L_1 \\ & \texttt{NEX} \\ L_1 : \ & \texttt{STEP } M \end{aligned}$$

If the received value v turns out not to fulfil the condition, the `NEX` instruction will abort this execution path and backtrack to some previously executed `STEP` M' instruction. In the same way, we can also undo assignments to variables (I2).

One of our key decisions during the design was to add an explicit notion of a *step* to our byte-code language, in form of the `STEP` M instruction. This allows us fine-granular control on the visibility of run-time effects from inside the virtual machine (I3), as no intermediate steps become visible unless said instruction is executed. This mechanism was also used to translate PROMELA's `atomic` and `d_step` blocks.

Requirement I4 postulates the avoidance of duplication when translating conditionals with multiple cases. An example translation would look as expected:

$$\begin{aligned} \mathcal{C}[\![\textbf{if } E_1 \textbf{ then } C_1 \textbf{ elsif } E_2 \textbf{ then } C_2 \textbf{ else } C_3 \textbf{ end}]\!] := \quad & L_1 : \ \mathcal{E}[\![E_1]\!] \\ & \qquad \texttt{JMPZ } L_2 \\ & \qquad \mathcal{C}[\![C_1]\!] \\ & \qquad \texttt{JMP } L_4 \\ & L_2 : \ \vdots \\ & L_3 : \ \mathcal{C}[\![C_3]\!] \\ & \qquad \texttt{JMP } L_4 \\ & L_4 : \end{aligned}$$

We assume here expressions E_i have no side effects and cannot block. A relaxation of these assumptions is also possible, with only a slightly more involved translation using the `ELSE` byte-code [89, pp.78ff].

The last two requirements, size of the formal framework (I5) and compositionality (I6), are harder to support based on conclusive arguments. We believe that we found a

good compromise for the complexity of our framework, and we base this assertion on the size of our implementation (Section 7.4) and the rapidness in which it was conceived. The compositionality of our approach remains to be evaluated.

7.6. Related Work

7.6.1. PROMELA Semantics

Several formal semantics for PROMELA have been proposed in the past, but it turns out that none of them covers all aspects of the language. The original publication [55] is incomplete in this sense and now partly outdated, as SPIN evolved. It was improved on by a more modular and less implementation-specific approach by Weise [96], but there the handling of nested `do` loops in combination with `goto` statements is unsound. Another incomplete attempt is from Bevier [10]. The specification is a Lisp program and as such peppered with implementation artefacts.

In contrast, our semantics is faithful to SPIN's PROMELA semantics. It mainly deviates in allowing nested scopes, in order to straighten out the rather confusing static semantics of declarations (variables can be used before being declared). Following our semantics, we developed a compiler for PROMELA, targeting the virtual instruction set defined in Section 7.1.3.

7.6.2. Virtual Machines

Virtual machines have been used extensively in Computer Science. A well-known example is for instance the work of Wirth on the Pascal programming language [97].

Independent to our work, two (unpublished, to the best of our knowledge) attempts of virtual machine models for restricted PROMELA-like languages have been brought to our attention [45, 88]. Geldenhuys [45] describes a virtual machine as part of the general design of a model checker, while our work is focused on providing a reusable component for state space generation.

ESML [32], the high-level language translated into byte-code is restricted in several ways when compared to PROMELA, and its underlying virtual machine inherits some of these restrictions. For example, it lacks support for asynchronous channels, shared variables and dynamic process creation.

Rosien [88, Section 8] describes some shortcomings of his attempt, for example the lack of arrays, no support for data types beyond integers, unclear semantics for `do` loops or handshake communication inside atomic blocks ("[...] causes undesired results, unexpected atomic deadlocks or otherwise erratic behavior.").

Besides that, we are in doubt that the architecture of Rosien's design can be adapted easily to e.g., distributed settings where successive states may be generated on different computers. This use case was specifically taken into account in the design of our VM.

Both papers do not provide a complete formal model of their VM or of the translation into their byte-code language, making it non-trivial to derive implementations from their work, neither are implementations readily available.

BACI

The *Ben-Ari Concurrent Interpreter* (BACI) suite in its latest version compiles a version of Wirth's Pascal enriched with concurrency constructs into the byte-code language PCODE, which is then executed on a virtual machine. BACI is widely used as teaching device for concurrency, not for verification purposes, hence the byte-code language is still relatively high-level and not stream-lined for simplicity and efficient execution. Also, the virtual machine does not allow to specify the granularity of visible actions, as is the case with ours.

To the best of our knowledge, formal semantics of the byte-code language or virtual machine are not available.

TyCO VM

Lopes at al. [70, 71], present a virtual machine for the process algebra of *Typed Concurrent Objects* (TyCO), a close relative to asynchronous π-calculus. Features include a process concept, communication channels, and a notion of atomic execution (coined *thread*). Since its virtual machine is meant for program execution rather than verification, it lacks nondeterminism, an external scheduler and invisible states, when compared to our work. Also, TyCO's more complex machine state is not designed for snapshotting and restarting. We stress the authors' report that their virtual machine executes efficiently, also due to optimizations carried out at byte-code level.

PROBMELA

A probabilistic extension of PROMELA is presented in [3]. Through private communication with one of the authors (Ciesinski) we recently learned about their endeavor to implement a virtual machine. No published work of these efforts is available so far, but the cited PROBMELA paper reveals a number of simplifying deviations from PROMELA semantics, e.g. atomic regions always running to completion, making them equal to PROMELA's `d_step` and thus obviating the need for priorities on byte-code level.

However, we see the existence of their project as evidence that we are on the right track, and we are confident that probabilistic extensions can be fitted into our virtual machine model. This is left as future work for a possible collaboration.

Java Path Finder and Bandera

Java Path Finder 2 [94] translates Java byte-code into *Bandera intermediate representation* (BIR), which then can be model-checked using Bogor [86], or translated to PROMELA, using SPIN as back-end model checker. The intermediate representation is a high-level guarded command language, not unlike PROMELA. While it can be translated further down to a certain extent, constructs like arrays, locks, exceptions, and high-level control constructs remain, complicating an implementation of its operational semantics. On the other hand, we are confident that BIR can be translated further down to an extended version of our byte-code language.

The Bogor framework consists of a large Java code base, which we conjecture is not easy to replicate in another language if needed. Again, from the tool point of view, our aim is not to beat the Bogor framework in terms of features, but rather to provide a small but versatile component which can easily be reused, or written from scratch based on a formal specification.

7.7. Conclusions

We presented a virtual machine-based approach to state-space generation, in which the virtual machine's instruction set doubles as intermediate language. The machine's semantics are straightforwardly implementable, thus encouraging reuse of our specification. Among the byte-code instructions are all operations commonly needed for the specification of concurrent systems: non-determinism, process creation, communication primitives, and a way to express scheduler constraints (atomic regions). As such, our byte-code language doubles as a general framework for the assignment of *executable* operational semantics to high-level modeling languages for concurrent systems.

Benchmarks showed that it is a usable alternative to SPIN in terms of speed, and superior for embedding into third-party model checkers. Although a *Just-in-time* compiler for our byte-code is conceivable to further increase the speed of our virtual machine, we believe the extra complexity is not worth the effort for now.

8. Conclusions and Future Research

In this thesis, we introduced a novel family of distributed algorithms for solving the model-checking problem for two fragments of Kozen's μ-calculus [60]: the alternation-free fragment L^1_μ, which subsumes the well-known Computation-Tree Logic CTL [40], and the fragment L^2_μ allowing formulas with one alternation, which subsumes other practically relevant logics, like Linear-Time Logic LTL [83] and CTL* [37].

Both our algorithms work within the same framework, namely Stirling's model-checking games [91], and we showed that the algorithm for the alternation-free fragment L^1_μ can be reused as subroutine for the case of one alternation. Depending on different needs, they can be tailored to exhibit on-the-fly properties.

We express the model-checking problem in terms of game graph which are subsequently colored by our algorithms. The coloring can then be translated back to provide an answer to the original question. Our key insight was to exploit the structure of μ-calculus formulas in order to find a partitioning of the game graph. This saves us from having to perform a cycle detection algorithm, which is considered prohibitively expensive in a distributed setting like ours.

Through experiments, we have shown that our algorithms are able to provide answers for very large models. In particular, we tested them on the largest transition systems of the *Very Large Transition Systems* (VLTS) benchmark suite which contains problems instances up to an order of 3×10^7 states and 1.5×10^8 transitions, inclusively.

The UppDMC implementation [52] of our algorithms was able to provide all of the missing results by utilizing the combined resources of a network of workstations, thus demonstrating that our algorithms can deal easily with real-world problems where sequential approaches fail. Furthermore, our experiments have shown that our algorithms scale well with increasing problem size.

In the second part, we turned our attention to the generation of state spaces suitable for consumption by model-checking algorithms, from a high-level description of model. We highlighted efficiency problems and issues of semantic complexity, and then proposed the translation to an intermediate representation in form of virtual-machine byte-code, which can be efficiently translated further into a low-level transition system representation, and optionally subjected to further optimizations. The usage of an intermediate representation is a widely employed trick within compilers to break down the complexity of a translation, and works just as well in the domain of state-space generation.

Additional benefits of our virtual-machine based approach also include its value as

encapsulated reusable component with a well-defined and concise interface, and the provision of executable operational semantics for modelling languages.

We also put our virtual-machine based approach to state-space generation under real-world scrutiny. We formalized and implemented a virtual machine expressive enough to suit as target for the translation of the modelling language PROMELA [48]. Benchmarks have shown that it is competitive in terms of speed with state-of-the-art tools, in some cases even outperforming SPIN [48].

Moreover, its additional advantage of allowing straightforward integration in distributed model-checkers has been confirmed by the integration with the DIVINE [35] framework for distributed model checking, yielding a distributed SPIN-compatible tool called DIVSPIN [69].

Future Work

Different Logics Probably an expectable future task is the extension of our distributed algorithms to handle higher alternation depths of the μ-calculus, thus providing more expressive power. Yet, we currently do not see a real need to go in this direction, as we already capture many temporal logics of practical interest, and it very fast becomes cumbersome to reason about the meaning of formulas with deeper nesting of alternating fixpoints.

Instead, we believe it is worthwhile to investigate into the opposite direction: identifying less expressive sub-logics, which lead us to specialized and even more efficient algorithms.

Distributed Fail-over While we now have at our disposal scalable distributed algorithms for verification of large systems, several research directions in this area have not been addressed much in the literature so far. In order to provide an industrial-grade solution, our algorithms would have to cope with exceptional situations, for example, in which one of the workstations of a NOW fails. Instead of terminating the whole computation, we would expect some kind of graceful degradation of service, possibly with some kind of fail-over mechanism. Such features become especially important if the number of processing units increases from tens to hundreds or thousands, as the probability of failure increases as well.

Grid Computing Although our algorithms are targeted at distributed computing environments, they still assume a high-speed private network to interconnect processors to offset the already high communication costs.

Grids, on the other hand, are geographically distributed computing resources connected via the Internet, for which such assumptions do not hold any longer. Conse-

quently, issues like load-balancing and more efficient ways to utilize the communication infrastructure become much more important for algorithms designed to run in a Grid context.

Virtual Machine Extensions For our virtual machine, we are looking into a language-independent byte-code optimization phase along the lines of Yorav and Grumberg's static analysis for state-space reductions [98]. This requires an extension of the language they considered. However we believe the careful choice of our virtual machine's primitive byte-code operations simplifies the formalization of their analysis considerably, and opens up possibilities for further optimizations.

Our virtual machine already provides a set of features suitable for the translation of high-level modelling languages (with SPIN as prominent example). However, we are also looking into extensions of our virtual machine with notions of time [93], probabilities [3], or dynamic memory allocation [64], and their effect on its complexity.

List of Algorithms

3.1. initializeConfiguration($conf$) 37
3.2. color($conf$) 37
3.3. colorizeComponent(Q_j), sequential version 38
3.4. color($conf$), top-down version 41
3.5. colorizeComponent$_i$(Q_j), parallel version 49
3.6. processSuccessors($conf, Q_j$), parallel version 49
3.7. color($conf$), parallel bottom-up version 50
3.8. recolorComponent$_i$(Q_j), parallel bottom-up version 50
3.9. Main procedure, parallel bottom-up version 50
3.10. color($conf$), parallel top-down version 53
3.11. recolorComponent$_i$(Q_j), parallel top-down version 53
3.12. Main procedure, parallel top-down version 54
3.13. L^2_μ-colorizeComponent(Q_j) 60

Bibliography

[1] H. R. Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 11 Apr. 1994.

[2] J. Augusto, M. Butler, C. Ferreira, and S. Craig. Using SPIN and STeP to verify StAC specifications. In *PSI'03*, number 2890 in LNCS, pages 207–213. Springer Verlag, July 2003.

[3] C. Baier, F. Ciesinski, and M. Größer. Probmela: a modeling language for communicating probabilistic systems. In *Proc. MEMOCODE*, 2004.

[4] J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE – Distributed Verification Environment. Submitted to PDMC'05's short presentations., 2005.

[5] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.

[6] J. Barnat, L. Brim, and J. Chaloupka. Distributed Memory LTL Model Checking Based on Breadth First Search. Technical Report FIMU-RS-2004-07, Faculty of Informatics, Masaryk University Brno, 2004.

[7] J. Barnat, L. Brim, and J. Chaloupka. From Distributed Memory Cycle Detection to Parallel LTL Model Checking. *Electronic Notes in Theoretical Computer Science*, 133(1):21–39, May 2005.

[8] S. Basonov. Parallel implementation of BDD on DSM systems. Master's thesis, Computer Science Department, Technion, 1998.

[9] A. Bell. *Distributed Evaluation of Stochastic Petri nets.* PhD thesis, RWTH Aachen, 2004.

[10] W. Bevier. Towards an operational semantics of PROMELA in ACL2. In *Proceedings of the 3rd International SPIN Workshop*, April 1997.

[11] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[12] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. volume 58 of *Advances in Computers*. Academic press, 2003.

[13] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation free μ–calculus. Technical Report AIB-04-2001, RWTH Aachen, Mar. 2001.

[14] B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 543–558. Springer, Apr. 2001.

[15] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*, volume 2318 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., 2002.

[16] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[17] P. Borovansky, C. Kirchner, H. Kirchner, P. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. In *Proc. of the First Int. Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[18] M. Bozga, S. Graf, and L. Mounier. If-2.0: A validation environment for component-based real-time systems. In K. L. Ed Brinksma, editor, *Proceedings of CAV'02 (Copenhagen, Denmark)*, volume 2404 of *LNCS*, pages 343–348. Springer-Verlag, July 2002.

[19] J. C. Bradfield. The modal mu-calculus alternation hierarchy is strict. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 233–246, Pisa, Italy, 26–29 Aug. 1996. Springer.

[20] L. Brim and J. Barnat. Distribution of Explicit-State LTL Model-Checking. In T. Arts and W. Fokkink, editors, *Electronic Notes in Theoretical Computer Science*, volume 80, pages 1–6. Elsevier, 2003.

[21] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model-checking based on negative cycle detection. In *Proceedings of 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, Lecture Notes in Computer Science. Springer, Dec. 2001.

[22] G. Cabodi, P. Camurati, and S. Que. Improved reachability analysis of large FSM. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.

[23] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 10(1):82–93, 1998.

[24] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. 2(3):279–287, Nov. 1999.

[25] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[26] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.

[27] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, 1996.

[28] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In *Proc. of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 153–173, 1995.

[29] R. Cleaveland and B. Steffen. A linear–time model–checking algorithm for the alternation–free modal mu–calculus. In K. G. Larsen and A. Skou, editors, *Proceedings of Computer-Aided Verification (CAV'91)*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58, Berlin, Germany, July 1992. Springer.

[30] L. A. Crowl. How to measure, present, and compare parallel performance. *IEEE Parallel & Distributed Technology*, 2(1):9–25, Spring 1994.

[31] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, Apr. 1994.

[32] P. de Villiers and W. Visser. ESML—a validation language for concurrent systems. pages 59–64. 7-th Southern African Computer Symposium, July 1992.

[33] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.

[34] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid, 1992.

[35] DiVinE. http://anna.fi.muni.cz/divine.

[36] D. D'Souza and M. Mukund. Checking consistency of SDL+MSC specifications. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2003.

[37] E. Emerson and J. Y. Halpern. 'Sometimes' and 'Not Never' revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1985.

[38] E. Emerson and C. Lei. Efficient model checking in fragments of the propositional μ–calculus. In *Symposion on Logic in Computer Science*, pages 267–278, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[39] E. A. Emerson. *Model checking and the mu-calculus*, volume 31 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, chapter 6. American Mathematical Society, 1997.

[40] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, Dec. 1982.

[41] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In C. Courcoubetis, editor, *Proc. 5th International Computer-Aided Verification Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 1993.

[42] T. M. P. I. Forum. Document for a Standard Message-Passing Interface. CS-93-214, University of Tennessee, 11 1993.

[43] H. Garavel and F. Lang. NTIF: A general symbolic model for communicating sequential processes with data. In D. Peled and M. Y. Vardi, editors, *FORTE*, volume 2529 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2002.

[44] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Pres, 94.

[45] J. Geldenhuys. Efficiency issues in the design of a model checker. Msc. thesis, University of Stellenbosch, South Africa, November 1999.

[46] S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. Amendola, and P. Marmo. A formal specification and validation of a critical system in presence of byzantine errors. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in Lecture Notes in Computer Science. Springer, 2000.

[47] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[48] J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors. *The Spin Verification System*, volume 32 of *DIMACS series*. American Mathematical Society, 1997. ISBN 0-8218-0680-7, 203p.

[49] O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 350–362. Springer, July 2001.

[50] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In O. Grumberg, editor, *Computer-Aided Verification, 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 20–35. Springer, June 2000.

[51] C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.

[52] F. Holmén, M. Leucker, and M. Lindström. UppDMC – a distributed model checker for fragments of the μ-calculus. In L. Brim and M. Leucker, editors, *Proceedings of the $3rd$ Workshop on Parallel and Distributed Methods for Verification*, volume 128/3 of *Electronic Notes in Computer Science*. Elsevier Science Publishers, 2004.

[53] G. J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. volume LNCS 1680, Toulouse, France, 1999. Springer Verlag.

[54] G. J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, Boston, MA 02116, September 2003.

[55] G. J. Holzmann and V. Natarajan. Outline for an operational-semantics definition of PROMELA. Technical report, Bell Laboratories, July 1996.

[56] A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–282, 1994.

[57] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 263–277, Pisa, Italy, 26–29 Aug. 1996. Springer.

[58] B. Jenkins. A hash function for hash table lookup. *Dr. Dobb's Journal*, September 1997.

[59] C. Joubert and R. Mateescu. Distributed local resolution of boolean equation systems. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'05 (Lugano, Switzerland)*. IEEE Computer Society Press, February 2005.

[60] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, Dec. 1983.

[61] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, Mar. 2000.

[62] M. Lange. Spielbasiertes Model-Checking für den alternierungsfreien mu-Kalkül. Master's thesis, Aachen, University of Technology, 1999. (German).

[63] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39, London, UK, 1999. Springer-Verlag.

[64] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 5th International SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1999.

[65] M. Leucker. Model checking games for the alternation free mu-calculus and alternating automata. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning "(LPAR'99)"*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 77–91. Springer, 1999.

[66] M. Leucker and T. Noll. Rewriting logic as a framework for generic verification tools. In *Proceedings of the Third International Workshop on Rewriting Logic and its Applications (WRLA'00)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[67] M. Leucker, T. Noll, P. Stevens, and M. Weber. Functional programming languages for verification tools: A comparison of ML and Haskell. *Software Tools for Technology Transfer*, 7(2):184–194, 2005.

[68] M. Leucker, R. Somla, and M. Weber. Parallel model checking for LTL, CTL* and L^2_μ. In L. Brim and O. Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier Science Publishers, 2003.

[69] M. Leucker, M. Weber, V. Forejt, and J. Barnat. DivSPIN – a SPIN compatible distributed model checker. Accepted for PDMC'05's short presentations, 2005.

[70] L. Lopes, F. Silva, and V. T. Vasconcelos. A virtual machine for the TyCO process calculus. In *PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, September 1999.

[71] L. Lopes and V. T. Vasconcelos. TyCO abstract machine — the definition. DCC 97–1, DCC-FC & LIACC, Universidade do Porto, May 1997.

[72] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technische Universität München, 1996.

[73] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[74] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[75] J. Meseguer. Rewriting as a unified model of concurrency. In *Proceedings Concur'90 Conference*, Lecture Notes in Computer Science, Volume 458, pages 384–400, Amsterdam, Aug. 1990. Springer. Also, Report SRI-CSL-90-02R, Computer Science Lab, SRI International.

[76] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer, 1993.

[77] A. A. Narayan, J. J. J. Isles, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned–roBBDs. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.

[78] D. Nicol, G. Ciardo, and J. Gluckman. Distributed state-space generation of discrete-state stochastic models. Technical report, Nov. 13 1995.

[79] P. P. P. Inverardi, H. Muccini. Automated check of architectural models consistency using spin. San Diego, California, 2001.

[80] R. Palmer and G. Gopalakrishnan. The parallel PV model checker. Technical Report FIMU-RS-2002-05, Masaryk University, Brno, Czech Republic, 2002.

[81] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.

[82] D. Peled. Ten years of partial order reduction. In *Proceedings of 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28, Vancouver, BC, Canada, 1998. Springer.

[83] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, Oct. 31–Nov. 2 1977. IEEE Computer Society Press.

[84] Z. Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[85] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, New York, 1982. Springer.

[86] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. *SIGSOFT Softw. Eng. Notes*, 28(5):267–276, 2003.

[87] S. H. Roosta. *Parallel Processing and Parallel Algorithms*. Springer, New York, Berlin, Heidelberg, 1999.

[88] M. Rosien. Design and implementation of a systematic state explorer. Msc. thesis, University of Twente, The Netherlands, March 2001.

[89] S. Schürmans. Ein Compiler und eine Virtuelle Maschine zur Zustandsraumgenerierung. Diplomarbeit, RWTH Aachen University, Oktober 2005.

[90] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In O. Grumberg, editor, *Computer-Aided Verification, 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer, June 1997. Haifa, Israel, June 22-25.

[91] C. Stirling. Games for bisimulation and model checking, July 1996. Notes for Mathfit Workshop on finite model theory, University of Wales, Swansea,.

[92] A. L. Stornetta. Implementation of an efficient parallel BDD package. Master's thesis, University of California, Santa Barbara, 1995.

[93] S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. In *Proceedings of TACAS '96*, volume 1055 of *LNCS*, 1996.

[94] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, 2000.

[95] M. Weber. Paralleles Model Checking. Master's thesis, Aachen, University of Technology, 2001. (German).

[96] C. Weise. An incremental formal semantics for PROMELA. In *Proceedings of the 3rd International SPIN Workshop*, April 1997.

[97] N. Wirth. Pascal-s: A subset and its implementation. In D. W. Barron, editor, *Pascal - The Language and its Implementation*, pages 199–259. John Wiley, 1981.

[98] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Form. Methods Syst. Des.*, 25(1):67–96, 2004.

[99] S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science*, pages 154–163, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

Symbols and Notations

$\leq$	partial-order relation	14
$\lessdot$	cover relation	14
Var	set of fixpoint variables	15
$Prop$	set of propositional variables	15
L_μ	set of μ-calculus formulas	15
$[\![K]\!]$	$\langle K \rangle$ or $[K]$ modality	15
$\#\varphi$	$\sigma X.\, \varphi$ or $[\![K]\!]\varphi$	15
Sub	set of subformulas	16
FV	set of free variables	16
BV	set of bound variables	16
$\mathcal{T}$	labelled transition system	17
V	valuation	17
ad	alternation depth	19
L_μ^n	fragments of L_μ	19
$Occ(\varphi)$	occurrence set of φ	20
$\rightarrow$	formula edge relation	20
$\mathcal{T}(\varphi)$	tree representation of φ	20
$\mathcal{G}(\varphi)$	graph representation of φ	21
$\Gamma(\mathcal{T}, \varphi)$	model-checking game	27
$\Rightarrow$	game move	27
$G^{\mathcal{T}}(s, \varphi)$	single play of a model-checking game	27
$\mathcal{GG}$	game graph	32
$\mathcal{Q}$	set of graph components	32
$\lfloor Q_i \rfloor$	set of escape configurations of Q_i	34
$\lceil Q_i \rceil$	set of initial configurations of Q_i	34
Γ	global machine state	86
Π	finite set of processes	86
Λ	local process state	88
$\rightarrow_{int}$	internal step transition	90
$\rightarrow_{step}$	intermediate transitions	94
$\xrightarrow{p}_{sched}$	scheduler transition	95

Index

μ-component, 23
ν-component, 23
ν-variable, 16
(strongly) connected component, 13

active, 88
algorithm
    distributed, 9
    enumerative, 8
    explicit-state, 8
    global, 7
    local, 7
    message-passing, 9
    parallel, 8
    symbolic, 8
alternation depth, 19
alternation-free, 2, 19
alternation-free fragment, 21
asymptotic run-times, 69

binary decision diagrams, 8
binder, 16
blocks, 46
boolean equation system, 10
bound, 16
bound variables, 16
bridge, 14

canonical decomposition, 34
channels, 89
combined complexity, 25
component, 13
component number, 24, 34
Computation-Tree Logic, 3, 15
configuration, 27
connected, 13
cooperative multitasking, 94
cover, 14
cover relation, 14
cycle, 13

data stack, 88
deadlock, 88
    global, 88
depends on, 19
determined, 31
digraph, 13
directed acyclic graph, 13
directed graph, 13

edges, 13
escape configurations, 34

fixpoint operators, 15
free, 16
free variables, 16

game
    parity-, 27
game board, 27
game graph, 9, 27, 31
game move
    Abelard, 28
    Eloise, 28
    existential, 27
    universal, 27
global state, 86
graph of φ, 20

graph representation, 20
guard, 94

Hasse diagram, 14
high-level language, 79
history-free, 30

inactive, 88
inherently sequential, 25
initial configurations, 34
initial state, 17
Initial stores, 88
interval, 14
is contained, 13

labelled transition system, 17
Linear-Time Logic, 3
local process state, 88
logic
    computation-tree, 3
    linear-time, 3

maximal, 14, 30
message complexity, 52
modalities, 15
model checking game, 27
model checking problem, 25
move, 27

Network Of Workstations, 1
Nick's Class, 25
nodes, 13
non-determinism
    conditional, 91
non-trivial, 13
normal, 17
NTIF, 82

occurrence set, 20
on-the-fly, 3
operational semantics
    executable, 3

partial order, 14
partially ordered set, 14
path, 13
path compression, 99
play, 27
player
    Abelard, 27
    Eloise, 27
poset, 14
process, 86
Process abstractions, 79
processor
    multi-core, 2
program complexity, 25

reached in, 13
reduction
    partial order, 97
rewrite engines, 81

scheduler transition, 95
sentence, 17
serialisation, 83
specification language compiler, 81
stable, 37, 42
state space
    explored, 99
state spaces, 75
statement merging, 99
stores, 88
strategy, 30
structural operational semantics, 81
subformulas, 16
subsumed, 19
succeeding positions, 25

terminal, 30
traceable, 80
transition
    intermediate-, 94
    scheduler-, 94

tree, 13
tree order, 14
tree representation, 20
type, 16

winner, 30
winning, 25
winning strategy, 30

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from

`http://aib.informatik.rwth-aachen.de/`.

To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities

2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java

2002-09 Markus Mohnen: Interfaces with Default Implementations in Java

2002-10 Martin Leucker: Logics for Mazurkiewicz traces

2002-11 Jürgen Giesl, Hans Zantema: Liveness in Rewriting

2003-01 * Jahresbericht 2002

2003-02 Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting

2003-03 Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs

2004-01 * Fachgruppe Informatik: Jahresbericht 2003

2004-02 Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting

2004-04 Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming

2004-05 Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.

Curriculum Vitae

Name	Michael Weber
Geburtsdatum	10.08.1976
Geburtsort	Düren

Bildungsgang

1987–1995	Franken-Gymnasium Zülpich Abschluss: Allgemeine Hochschulreife
1995–2001	Studium der Informatik an der RWTH Aachen Abschluss: Diplom
2001–2005	Wissenschaftlicher Angestellter am Lehrstuhl für Informatik II (Prof. Dr. Klaus Indermark), RWTH Aachen
seit Nov. 2005	Postdoc am Centrum voor Wiskunde en Informatica Amsterdam, Niederlande