Enhancing State Space Reduction Techniques for Model Checking

Dragan Bošnački
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2001-14

Printed by University Press Facilities, Eindhoven
ENHANCING STATE SPACE REDUCTION TECHNIQUES FOR MODEL CHECKING

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. R.A. van Santen, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 7 november 2001 om 16.00 uur

door

DRAGAN BOŠNAČKI

geboren te Berovo, Macedonië
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J.C.M. Baeten  
en  
prof.ir. M.P.J. Stevens

Copromotor:

dr. D.R. Dams
Acknowledgments

There are many people and institutions who helped me in various ways during the writing of this thesis. Aware of the inevitable risk of forgetting someone, I will try to list them with some words of gratitude that unfortunately often will be only a pale shadow of what they should have been. For those that I have forgotten to mention – a proverb from my homeland: "May the evil also forget you".

First of all, I would like to thank my copromotor Dennis Dams. I consider myself extremely fortunate that I had during all these years his valuable guidance and friendship. His was always able to help with all my technical questions, as well as to give me the right advise at the right time. Dennis is also a coauthor of several papers whose modified versions became chapters of this thesis.

No less gratitude is reserved for my promotor Jos Baeten. He opened for me the doors of the Western European science by inviting me to a short stay at the Eindhoven University of Technology in the fall of 1996. Since then he has been offering me continuously technical and moral support. He also deserves the credit for creating a nice working atmosphere in the Formal Methods group. Jos is the only boss that I know that regularly goes from door to door in order to gather his group for lunch.

Gerard Holzmann provided for me the opportunity to spend an unforgettable summer 1998 in Bell Laboratories. The experience of walking in the corridor with probably the greatest concentration of computer science celebrities in the world is something I always talk about with pleasure. Thanks go to Gerard also for patiently answering numerous technical questions regarding Spin and model checking in general.

Most of the work on this thesis was done under the auspices of the European Community’s Fourth Framework project Verifying Industrial REactive Systems (VIRES) (Esprit LTR 23498) which I gratefully acknowledge. I would like to also thank all the participants in the project with whom I had the opportunity to discuss my research.

During the Ph.D. studies in Eindhoven I was on leave from the Institute of Informatics, Faculty of Natural Sciences and Mathematics of the St. Cyril and Methodius University in Skopje, Macedonia. I am immensely grateful to my colleagues from the Institute for their understanding and support.

On this occasion I should not forget the technical and financial support by the IPA postgraduate school. The events organized by the school offered an excellent opportunities for learning and enjoyment.

I am grateful to Leszek Holenderski for coauthoring the papers who became basis for two of the chapters in this thesis. We spent a lot of time on interesting discussions, often outside our research. We had different opinions about many issues, but at the end I think that I learned many interesting things from him.
Tanks go to Twan Basten for coauthoring of Chapter 2, numerous discussions on the topic of partial order reduction, and helping me with the Samenvatting in Dutch.

Natalia Sidorova is a coauthor of the paper who is a basis of Chapter 5 of the thesis. I am also grateful to her for the numerous discussions regarding the verification of timed systems.

I would like to thank Mario Stevens for accepting to be my second promotor.

Doron Peled and Hans Toetenel kindly accepted to be members of the reading committee and gave valuable insightful comments on the first version of this document, for which I am indebted to them.

I am mostly grateful to Smile Markovski for his support during the years I spent at the Institute of Informatics in Skopje, Macedonia. Many things in theoretical computer science I learned from him. I am also honored that he accepted to be a member of the defense committee for my thesis.

Jan-Friso Groote and Antti Valmari also kindly accepted to be members of the defense committee, for which I honor them.

Many thanks to Tijn Borghuis and Tim Willemse for their helpful remarks on parts of the manuscript and their friendship.

I am grateful to Peter Hilbers for believing in my potentials as a researcher and his understanding during the finishing of the last chapter of the thesis.

Rob Gerth and Bart Knaack drew my attention to Spin, for which I would like to thank them. Rob also played a significant role in my decision to come to Eindhoven.

Many people from the University helped indirectly with my thesis, for which I give them my warmest thanks: Suzana Andova for her moral support, constructive criticism and helping me not to forget the taste of the Macedonian cuisine; Marcella de Rooij, Victor Bos, Georgi Jojgov and Martijn Oostdijk for being with me during my ups and downs and letting me winning couple of Risk games; Sjouke Mauw for his friendship, valuable advises and always supportive smile; Ana Sokolova, Anne-Meta Overstegen, Ruurd Kuiper, Rob Nederpelt, Nicu Goga, Peter Peters, Roel Bloo, Michael Franssen, Andre Engels, Erik de Vink, Kees Middelburg, Loïc Feijs, François Monin, Francien Dechesne, Dmitri Chkliiaev, Elize Russell, Michel Reniers, Sergey Tourine, Nino Mushkudiani, Olga Caprotti, Oana Balan, Marcin Grodek, Martin Hill, Desiree Mijers, Jolande Mathijsse, Trees Klaver, and Monique Bechtold for making my days at the University more enjoyable.

The successful finishing of this thesis would not be possible without the good education I received at the St. Cyril and Methodius University in Skopje, Macedonia, during the postgraduate studies at the Institute of Informatics, Faculty of Natural Sciences and Mathematics, and the undergraduate studies at the Faculty of Electrical Engineering.

The last, but by far the greatest gratitude goes to my parents for their endless love, all kinds of unconditional support, and for always believing in me much more than I believed in myself.

Contents

Acknowledgments .......................................................... iii

Contents ................................................................. v

1 Introduction .......................................................... 1

2 Enhancing Partial-Order Reduction via Process Clustering .... 13

3 Partial Order Reduction in Presence of Rendez-vous
   Communications with Unless Constructs and Weak Fairness .... 31

4 Integrating Real Time into Spin: A Prototype Implementation 55

5 Model Checking SDL with Spin ................................. 77

6 A Heuristic for Symmetry Reductions with Scalarsets ........ 93

7 Nested Depth First Search Algorithms for Symmetry Reduction
   in Model Checking ................................................ 123

8 Conclusions .......................................................... 169

9 Samenvatting ......................................................... 173
1. Introduction
Introduction

1 Motivation and Background

1.1 The Need for Formal Methods

One can often hear the expression “with computer precision” which is usually used to describe a flawless impeccable execution of some task. Unfortunately, computers do not always work correctly. Moreover, the errors of software and hardware systems applied in safety-critical systems like nuclear power plants, medical equipments, highway and air traffic control, railways, electronic commerce, can be potentially disastrous. Such grim predictions are supported by famous failures, like the Ariane 5 space rocket disaster or the Pentium bug, costing billions of dollars [3]. But as information technology penetrates in our daily life, there will be even more prosaic reasons why the errors will not be tolerated. Just imagine a football aficionado whose TV crashes during the penalty shootouts because of the embedded software, or you being cut by your intelligent shaver. There are predictions that in the future the main problem for the application of information technology will not be the lack of raw computational power, but our inability to develop complex systems with sufficient confidence in their correctness.

The traditional engineering techniques for validation, like peer review, simulation, and testing, have often proved inadequate and too expensive to avoid errors in information processing artifacts. One reason for this is that they explore only a part of the possible behavior of the system. As a result some erroneous behavior often escapes undetected.

In the last two decades, alternative, formal approaches for validation have emerged. They are based on formal methods, i.e., on methods that are more systematic and that have solid mathematical foundations. One of the main advantages of formal methods is that they perform exhaustive exploration of all possible behaviors. In this thesis we concentrate on model checking as a formal approach for debugging and verification of hardware and software.

1.2 Model Checking

Roughly speaking, model checking [3] is an automated technique that, given a model of the system and some property, checks whether the model satisfies the property. Originally suggested in the beginning of the eighties independently by Emerson and Clarke, and Quelle and Sifakis, nowadays the technique gains popularity, with several major companies developing in-house model-checking tools. Compared to the other (semi)automated formal techniques (for instance, deductive methods, like theorem provers) model checking is relatively easy to use. The specification of the model is very similar to programming and as such
it does not require much additional expertise from the user. The verification procedure is completely automated and often takes only several minutes. Another important advantage of the method is that, if the verification fails, the possible erroneous behavior of the system can be reproduced. This significantly facilitates the location and correction of the errors.

**The Characteristics of the Analyzed Systems.** In this document we deal with *concurrent* systems, i.e., systems composed of components that can operate *concurrently* and *communicate* with each other. It is assumed that the (components of the) systems are *reactive*, i.e. continuously interact with their environment. Typical representatives of reactive systems are communication protocols, operating systems, process control software, and aircraft control. This is in contrast with traditional sequential programs which can be seen as data transformers with single input and single output, i.e., interacting with the environment only at their beginning and end. Concurrency is modeled by *interleaving*, i.e., it is assumed that only one component executes an action at a time and the concurrent actions are arbitrarily ordered. Considering the reactive nature of the concurrent systems that are analyzed, we are interested in the verification of *control (interaction) properties*, rather than properties that are related to data.

**The Formal Framework and the Scope.** As outlined in [16], each framework for formal analysis consists of four components:

- *formal semantics* in which the system and the property are interpreted;
- *formal language* for describing the system;
- *formal language* for describing the property;
- *formal, preferably automated, techniques* with which it can be checked whether the system satisfies the property;

We use variants of *(labeled) transition systems* to give the semantics of the systems and properties. In other words, the semantics of the system is its state space, i.e., the set of all possible states that the system can reach and the transitions between them. The state transition systems are represented in a natural way as labeled graphs. Model checking by definition is applied on systems with finite state space.

Although some of the presented research is adaptable to the branching-time framework, we mainly work with trace semantics, i.e., we specify the properties either directly in the model, or in *linear temporal logic (LTL)* [6] and, more general, *Büchi automata* [19].

The results presented in the thesis are derived using directly the semantic model, i.e., the labeled transition systems, and as such are independent of the language in which the (model of the) system is specified. Having said this, we also emphasize that most of the work presented in this document was instigated by and implemented in the model checker *Spin* with its input language *Promela*, developed by Gerard Holzmann at Bell Laboratories [11].
Further, we consider model checking algorithms based on explicit enumeration of the state space, as opposed to the symbolic algorithms based on binary decision diagrams (c.f. [3]), for instance.

Although it is safe to conjecture that at least some of our results can be used for hardware verification, a tacit assumption throughout the thesis is that we are targeting debugging and verification of software systems.

The State Space Explosion Problem. Model checking requires search of the state space of the system, which may increase exponentially with the size of the system description. As a consequence, one of the major bottlenecks in model checking is the so called state space explosion [22]. The state space explosion is simply the combinatorial explosion caused by the interaction (interleaving) of the components and/or by the usage of data structures ranging over many different values. There exist numerous techniques to combat this problem, like symbolic verification, on-the-fly verification, abstraction, partial order reduction, symmetry reduction, etc. In our research we put emphasis on the last two: partial order reduction and reduction based on symmetry.

2 Contributions of This Thesis

The main contributions of this thesis are several improvements of the techniques for state space reduction. As stressed above, we focus on partial order reduction and reduction based on symmetries. The basic idea behind both techniques is to restrict the part of the state space which is explored for the verification in such a way that the properties of interest are preserved. However, in general they exploit different features of the concurrent systems and as a consequence they use different algorithms. In the first two parts of this section we briefly give the intuition behind each of the techniques and summarize the corresponding results.

The practical component of our contribution is reflected in the implementation of almost all of the obtained theoretical results. In this context we developed several upgrades of the model checker Spin and wrote accompanying programs. The prototype implementations were successfully tested on case studies from the academic literature and on protocols originating from industry. The last part of the section gives a brief overview of these practical aspects.

2.1 Partial Order Reduction

Partial order reduction [9, 18, 21] exploits the independence of the checked property from the execution order of the statements in the program (model description). More specifically, two statements $a$, $b$ are allowed to be permuted precisely when, if for all sequences $v$, $w$ of statements: if $vabw$ (where juxtaposition denotes concatenation) is an accepted behavior, then $vbaw$ is an accepted behavior as well. In a sense, instead of checking all the execution sequences, the desired property is checked only on representative sequences, which results in
significant savings in space and time. Finding the optimal relation for independence (permutability) of statements can be as difficult as the original verification problem [3]. Consequently, in practice only sufficient conditions for such a permutability are used that can be checked locally and preferably using only syntactic criteria, i.e., directly from the system specification. The actual reduction of the state space is realized during the state space exploration by limiting the search from a given state $s$ to only a subset of the statements that are executable in $s$.

Among the results that we obtained regarding partial order reduction, we emphasize the following:

**Partial order reduction for discrete time.** We present an adaptation for timed systems of the untimed algorithm for partial order reduction by Peled and Holzmann [18, 12], under the assumption that time is modeled with integers. One can consider that partial order reduction techniques consist of two parts. The first part is related with the determining the independence relation between statements. As we mentioned above, this part is usually done before the state space exploration starts. The second part is the actual exploration algorithm which, based on the independence relation and the structure of the state space, has to chose the subset of transitions which have to be explored from each state. Our main idea regarding the discrete-time extension is related to the criteria for independence of the statements (actions) which are applied on timers. As a consequence, the adaptation is independent of the rest of the engine of the partial order algorithms, which means that the proposed extension can be easily adapted to other approaches to partial order reduction, like for instance [9] or [21]. The implementation of the algorithm in the extension of Spin with discrete time, DT Spin, showed encouraging results (see also the discussion about DT Spin in the Tools section below).

**Compatibility of partial order reduction for systems in which synchronizing (rendez-vous) communications are combined with priority choice and/or weak fairness.** When synchronizing (rendez-vous) communications are used in Promela models, the priority choice construct `unless` in general is not compatible with partial order reduction algorithms. Also the weak fairness algorithm in Spin is not compatible with partial order reduction in presence of rendez-vous statements. Interestingly enough, it turns out that the same erroneous reduction pattern causes the incompatibility in both cases. After identifying this pattern we propose several solutions such that the power of partial order reduction can be retained in the presence of unless and weak fairness.

It can be expected that the importance of the above mentioned compatibilities will grow in the future, especially for the priority choice (`unless`). Namely, the `unless` statement in Promela is a natural way to model exceptions. With the popularity of the latter as a concept in modern programming languages (like Java), the compatibility of the priority choice with the other features of verification tools is an important advantage.
Exploiting system hierarchy for partial order reduction. Most of the model checking techniques treat the model as a flat composition of processes. Our approach of partial order reduction via process clustering shows how one can profit from the model structure for better state space reduction. Usually, the heuristics for independence of statements in partial order reduction regard a statement as either global or local, depending on the scope of the objects that are accessed by the statement. Our main idea is to introduce instead of this two-level view, several levels of globality of statements based on the structural hierarchy of the system. Most of the time this hierarchy is obtained for free, i.e., it is already contained in the specification (for instance, imposed by the syntax of the specification or modeling language). We show that employing a more refined notion of globality results in significant reductions (sometimes several orders of magnitude) of the state space, compared to the case when the standard two-level approach is used. Moreover, the gain in state space is without significant penalties in the verification time.

2.2 Symmetry Reduction
Symmetry based state space reductions [14, 15] exploit the inherent symmetry of the analyzed system. The main idea behind the technique can be illustrated on variants of the mutual exclusion problem. Assume that we have two processes, $A$ and $B$. Then, for the verification of the mutual exclusion property, the state in which process $A$ is in the critical section, while process $B$ can enter the critical section (and violate the mutual exclusion property) is equivalent to the symmetric state where the roles of $A$ and $B$ are swapped. More formally we say that the states are equivalent under permutation (in this case a simple swap) of the process IDs. The symmetric states are grouped in equivalence classes. Whenever during the exploration of the state space a state is generated which is the same up to a permutation of process IDs (i.e., belongs to the same equivalence class) as an already visited state, the search can be pruned. Formally speaking, we can consider that instead of exploring the original (concrete) state space, we check an abstract state space whose states are (representatives of the) symmetry equivalence classes of states from the original state space.

Below we give our main contributions related to symmetry based reductions.

Developing efficient heuristics for finding representatives of equivalence classes. The problem of finding (canonical) representatives of the equivalence classes under symmetry is equivalent to the graph isomorphism problem for which no polynomial solution is known. As a result the gain in state space can be diminished by unacceptably long verification times. We propose four versions of a new heuristic for finding representatives. The implementation in Spin showed that the heuristics work well and can potentially significantly improve the performance of verification tools.

Model checking under weak fairness using symmetry reductions. We present an algorithm for model checking that combines weak fairness with sym-
The algorithm is based on the nested depth-first search algorithm (NDFS) by Courcoubetis, Vardi, Wolper and Yannakakis [4]. This is in contrast with similar algorithms that exist in the literature which require finding maximal strongly connected components (MSCC) in a graph. As a consequence, our algorithm has all the advantages that NDFS has over the MSCC approach. It is compatible with the approximative verification techniques of [11, 24], which is not the case with the MSCC based algorithms. Also, we argue that in practice our algorithm has better time and space complexity. Finally, with our algorithm it is easier to reconstruct a diagnostic execution sequence which leads to a possible error.

As intermediate results we give an NDFS based algorithm for state space reduction techniques that preserve bisimulation [7]. (Symmetry reduction is a special case of the latter.) Also we discuss a modification of Spin’s weak fairness algorithm (without symmetry). To this end we introduce the notion of weakly fair extension of transition system which facilitates the correctness proof of the algorithm.

2.3 Tools and Case Studies

One of the main advantages of the model checking techniques is that they are readily implementable in tools. A significant part of the work of this thesis was spent on the various extensions and improvements of model checking tools and accompanying programs. Almost all theoretical results listed above have been implemented and the prototype implementations have been evaluated on case studies. As mentioned already, most of the practical work is related to the model checker Spin, which is one of the most popular model checking tools, successfully used in academia and industry. Below we give a short summary of the developed or upgraded software.

**DT Spin.** Standard Spin does not feature the possibility to express time quantitatively. For instance, given two consecutive statements $a$ and $b$, we know that $b$ will be executed after $a$, but we cannot specify, for instance, that $b$ will happen five time units after $a$. This can be a handicap when dealing with systems that critically depend on timing. **DT Spin** is an extension of Spin with discrete time which allows the specification of such quantitative timing relations between statements. DT Spin is fully compatible with Spin and allows verification of all the properties that can be verified with standard Spin. Moreover, the above mentioned discrete-time partial order reduction algorithm is implemented in DT Spin, which resulted in significant savings in the state space in the case studies.

**if2pml** is a translator from the language IF into Promela, the input language of Spin. IF [2] is an emerging language designed at VERIMAG Grenoble that aims at providing a common intermediate format for connecting various formal languages and tools. The translator if2pml was written as a second part of a
translator from the language SDL to Promela. It was successfully used in the validation of MASCARA, a communication protocol developed by industry, whose original specification was written in SDL. During the development of if2pml we came across several non-trivial problems that had to be resolved, most of them related to the timing features of IF and SDL.

SymSpin is an extension of Spin for exploiting symmetry based reduction techniques. More particularly, we have implemented on top of Spin the above mentioned heuristics for finding representatives of the equivalence classes. In most of the case studies the obtained reduction approached the theoretical limit of $N!$, where $N$ is the number of processes in the system.

3 Related Work

Related work is discussed in more detail in each chapter. Here we briefly outline the most important related references grouped according to the subject.

Partial order reduction Our work on partial order reduction is mostly based on the partial order algorithm by Doron Peled [18] and on its variant (a joint work with Gerard Holzmann) which is implemented in Spin [12].

Although there exist several attempts to extend the partial order reduction techniques to timed systems (e.g. [5]) with dense time (i.e., time is modeled with real numbers), to the best of our knowledge there is no successful implementation of any of them. We are not aware of any work on partial order reduction for discrete time (time modeled as integers).

It was already mentioned that partial order reduction algorithms during the state space search explore only a subset of the transitions which originate in a given state. The algorithm of [18, 12] requires that all transitions in this subset belong to the same process. Our improvement based on process clustering removes this constraint. We only require that the transitions that are in the subset belong to a same group (cluster) of processes. There are several references in the literature that deal with grouping actions of several processes together for the explored subset. (Actually, one of the very first references in partial order reduction [17] is based on this kind of idea.) However, to the best of our knowledge there are no papers which do this grouping by exploiting the hierarchical structure of the system. Our paper was inspired by the NEXT reduction heuristic from [1]. As shown in [1], the NEXT heuristic in general capitalizes on different aspects of concurrency in order to reduce the state space, and as such differs from partial order reduction.

To the best of our knowledge, the compatibility of partial order reduction with fairness and priority choice in presence of synchronizing communication has not been treated in the literature before.
Symmetry reduction Probably the most referenced work regarding symmetry reductions is that of Ip and Dill [14]. We also rely mainly on this work, particularly in our paper about the heuristics for finding representatives. In a sense, one can consider our work as complementary to [14], where the main attention is devoted to the question how to efficiently detect the symmetries in the system, i.e., how to ensure that the obtained state space is symmetric. On the other hand, we deal with the question how to detect the symmetry between the states, i.e., how to detect during state space exploration that two states are equivalent, provided that the state space is symmetric.

The problem of finding representatives is (most of the time in very general terms) discussed in [14, 8, 15]. In our opinion the heuristics we propose have advantage in their simplicity and efficiency.

The only algorithms for reconciling symmetry reduction and weak fairness that we could find in the literature were the ones from [8] and [10]. As mentioned above, our algorithm has some advantages over these two algorithms because it is compatible with approximative verification techniques, in practice requires less time and memory, and has better diagnostic output in case an error is detected.

Verification of SDL models with Spin We are aware of at least two other attempts to verify SDL with Spin [13, 20]. In our opinion they tackle successfully only the untimed part of the SDL specification. The main advantage of our approach is that it also correctly captures the quantitative aspects of the behavior related to SDL timers.

4 Organization of the Thesis

This document is mostly a collection of articles whose original versions already have been published or submitted for publication. Consequently, they are meant to be self contained and are independent of each other. Therefore, some overlap and repetition of the basic concepts is unavoidable.

The thesis can be roughly divided into three parts. The first part contains the papers which describe the contributions related to partial order reduction techniques. The second part is the paper on verification of SDL specifications with Spin, containing the main case study, the MASCARA protocol. The last part comprises two papers related to the symmetry based reductions.

The first part consists of Chapters 2, 3 and 4. Chapter 2 discusses the exploitation of the system hierarchy for more efficient partial order reduction. In Chapter 3 we give a solution for the compatibility of partial order reduction in models that contain synchronizing (rendez-vous) communication statements, with priority choice and/or weak fairness. The main subject of Chapter 4 is the extension of Spin with discrete time and an adaptation of the untimed algorithm for partial order reduction.

Chapter 5 describes the verification of SDL specifications with the model checker Spin. In that context the developed methodology and tools, among which the translator if2pml, are described.
The third part consists of Chapters 6 and 7. Chapter 6 deals with the heuristics for efficiently finding representatives of equivalence classes. Chapter 7 presents the algorithm for model checking under weak fairness using symmetry reduction.

The last Chapter 8 concludes the thesis with a summary of the results and a discussion of the directions for future work.

4.1 The Origin of the Chapters

Almost all the work on this thesis was done under the auspices of the European Community’s Fourth Framework project *Verifying Industrial REactive Systems (VIRES) (Esprit LTR 23498)* [23]. Most of the material that is presented in this document has been published or submitted for publication. We give below the origin of each of the chapters.

Chapter 2 is the paper


A previous version of the paper appeared as


Chapter 3 is a revised version of the paper


Chapter 4 is a combination of revised versions of the papers


and


Chapter 5 is the paper
Chapter 6 is a combination of revised versions of the papers:


and


Chapter 7 is unpublished so far.

References

2. Enhancing Partial-Order Reduction via Process Clustering

This chapter is an extended version of the paper

T. Basten, D. Bošnački,
*Enhancing Partial-Order Reduction via Process Clustering,*
16th IEEE Conference on Automated Software Engineering ASE 2001,

A previous version of the same paper also appeared as

T. Basten, D. Bošnački,
*Enhancing Partial-Order Reduction via Process Clustering,*
Enhancing Partial-Order Reduction via Process Clustering

Twan Basten\(^1\) and Dragan Bošnački\(^2\)

\(^1\) Dept. of Electrical Eng., Eindhoven University of Technology,
PO Box 513, NL-5600 MB Eindhoven, The Netherlands
e-mail: a.a.basten@tue.nl

\(^2\) Dept. of Computing Science, Eindhoven University of Technology
PO Box 513, NL-5600 MB Eindhoven, The Netherlands
e-mail: dragan@win.tue.nl

Abstract. Partial-order reduction is a well-known technique to cope with the state-space-explosion problem in the verification of concurrent systems. Using the hierarchical structure of concurrent systems, we present an enhancement of the partial-order-reduction scheme of [12, 19]. A prototype of the new algorithm has been implemented on top of the verification tool SPIN. The first experimental results are encouraging.

1 Introduction

Over the last decades, the complexity of computer systems has been increasing rapidly, with a tendency towards distribution and concurrency. The correct functioning of these complex concurrent systems is becoming an ever larger problem. Many verification and proof techniques have been invented to solve this problem. An important class of techniques are those based on a fully automatic, exhaustive traversal of the state space of a concurrent system. Well-known representatives are the various model-checking techniques.

An infamous problem complicating an exhaustive traversal of the state space of a concurrent system is the state explosion, caused by the arbitrary interleaving of independent actions of the various components of the system. Several techniques have been developed to cope with this problem. Partial-order reduction is a very prominent one (see, for example, [1, 7, 8, 11, 12, 18–22]). It exploits the independence of actions to reduce the state space of a system while preserving properties of interest. During the generation of a state space, in each state, a subset of the enabled actions satisfying certain criteria is chosen for further exploration. Following [12, 19], we call these sets *ample* sets.

The traditional approach to partial-order reduction (see, for example, [12, 19]) deals with systems seen as an unstructured collection of sequential processes running in parallel. However, many systems have inherent hierarchical structure, imposed either explicitly by the language used for the system specification that groups processes in blocks or similar constructs (e.g., SDL, UML), or implicitly by the interconnections among processes. In this paper, we present a partial-order-reduction algorithm that exploits the hierarchical structure of a system.
As our starting point, we take the partial-order algorithm of Holzmann and Peled [12, 19]. This algorithm is implemented in the verification tool SPIN [9] and has proven to be successful in coping with the state-space explosion. It is also sufficiently flexible to allow extensions (see for instance the extensions for timed systems in [4, 17]). The algorithm uses a notion of safety to select ample sets. The safety requirement is imposed via syntactical criteria to avoid expensive computations during the state-space traversal. These criteria yield ample sets consisting of either all enabled actions of a single process or all enabled actions of all processes. Our idea is to introduce a gradation of the safety requirement based on the hierarchical structure of a system. To this end, we introduce the concept of a clustering hierarchy to capture the system hierarchy and the induced (in)dependencies among processes. This generalization allows ample sets consisting of actions from different, but not necessarily all, processes. Our cluster-based algorithm is a true generalization of the partial-order-reduction algorithm of [12, 19]. It can also be seen as a version of the algorithm of Overman [18], adapted for clustering hierarchies and LTL model checking. We implemented our algorithm on top of the verification tool SPIN. The results obtained with the prototype are encouraging, in particular, considering that a visual language is being developed for SPIN [15] that combines naturally with our cluster-based reduction algorithm.

The remainder of this paper is organized as follows. Section 2 explains the state-space-explosion problem and the basic concepts that play a role in this paper. Section 3 presents some known theoretical results on partial-order reduction as well as the reduction algorithm of [12, 19]. In Section 4, we present our cluster-based partial-order-reduction algorithm. Section 5 gives experimental results. Finally, Section 6 contains concluding remarks.

Acknowledgments. Our work is inspired by the Next heuristic of Rajeev Alur and Bow-Yaw Wang presented in [2]. We thank Dennis Dams and Jeroen Rutten for their contributions.

2 Preliminaries

State spaces of concurrent systems. Concurrent systems typically consist of a number of processes running in parallel. To exchange information, these processes communicate via messages and/or shared memory. The left part of Figure 1 shows the very simple concurrent system example. It consists of three processes, P0, P1, and P2, each one executing a sequence of two actions. Assuming that there is no communication and, thus, all actions can be executed independently, the right part of Figure 1 shows the state space of system example.

The numbers 00 through 08, 10 through 18, and 20 through 28 represent states of system example. The states encode all relevant information of example such as, for example, the values of variables and local program counters. The initial state of the system is state 00 (marked with a small arrow). The labeled arrows in Figure 1 correspond to state changes or transitions of the system. The labels link transitions to actions of the system. For example, action labels a0,
and \(b_1\) correspond to actions \(a_0\) and \(b_1\) of system \textit{example}. Note that parallel arrows in Figure 1 are assumed to have identical action labels.

The example of Figure 1 illustrates a well-known problem complicating the verification of concurrent systems. Clearly, each of the processes in \textit{example} can only be in three different states. However, Figure 1 shows that the complete state space of \textit{example} consists of 27 (=\(3^3\)) states. The state space of a system with a fourth process (executing two actions) already consist of 81 (=\(3^4\)) states, whereas the state space of a system consisting of two processes consists of only 9 (=\(3^2\)) states. This example illustrates that the state space of a concurrent system grows (in the worst case) exponentially if the number of processes in the system increases. This phenomenon is known as the state-space explosion. Obviously, the state-space explosion complicates verification techniques that are based on an exhaustive traversal of the entire state space of a concurrent system.

**Labeled transition systems.** The notion of a state space is the most important concept in this paper. To formally reason about state spaces, we introduce the notion of a \textit{labeled transition system}.

**Definition 1 (Labeled transition system).** A \textit{labeled transition system}, or simply an \textit{LTS}, is a 6-tuple \((S, \hat{s}, A, \tau, \Pi, L)\), where

- \(S\) is a finite set of states;
- \(\hat{s} \in S\) is the initial state;
- \(A\) is a finite set of actions;
- \(\tau : S \times A \rightarrow S\) is a (partial) transition function;
- \(\Pi\) is a finite set of boolean propositions;
- \(L : S \rightarrow 2^\Pi\) is a state labeling function.

Consider again the example of the previous paragraph. The states, the initial state, the actions, and the transition function of an LTS for \textit{example} are easily derived from the right-hand side of Figure 1. However, the figure does not contain any labeling of states with propositions. Propositions and state labeling are included in the definition of an LTS because they play a role when one is
interested in verifying specific properties of a concurrent system. Before explaining these last two elements of an LTS in some more detail, we introduce some auxiliary notions. Let $T = (S, \hat{s}, A, \tau, \Pi, L)$ be some LTS.

An action $a \in A$ is said to be enabled in a state $s \in S$, denoted $s \overset{a}{\to}$ iff $\tau(s, a)$ is defined. The set of all actions $a \in A$ enabled in a state $s \in S$ is denoted $\text{enabled}(s)$; that is, for any $s \in S$, $\text{enabled}(s) = \{ a \in A \mid s \overset{a}{\to} \}$. A state $s \in S$ is a deadlock state iff $\text{enabled}(s) = \emptyset$.

In the previous paragraph, the notion of a transition has already been mentioned. Formally, transition function $\tau$ of LTS $T$ induces a set $T \subseteq S \times A \times S$ of transitions defined as $T = \{ (s, a, s') \mid s, s' \in S \land a \in A \land s' = \tau(s, a) \}$. To improve readability, we write $s \overset{a}{\to} s'$ for $(s, a, s') \in T$. Besides the number of states of a concurrent system, also the number of transitions of the system is a factor complicating verification. Note that the LTS of Figure 1 contains 54 transitions.

An execution sequence of LTS $T$ is a (finite) sequence of subsequent transitions in $T$. Formally, for any natural number $n \in \mathbb{N}$, states $s_i \in S$ with $i \in \mathbb{N}$ and $0 \leq i \leq n$, and actions $a_i \in A$ with $i \in \mathbb{N}$ and $0 \leq i < n$, the sequence $s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \ldots s_{n-1} \overset{a_{n-1}}{\to} s_n$ is an execution sequence of length $n$ of $T$ iff $s_i \overset{a_i}{\to} s_{i+1}$ for all $i \in \mathbb{N}$ with $0 \leq i < n$. State $s_n$ is said to be reachable from state $s_0$. A state is said to be reachable in $T$ iff it is reachable from $\hat{s}$. The LTS of Figure 1 has 90 execution sequences of length six all starting from the initial state and leading to deadlock state 28.

**Properties of concurrent systems.** There are many different kinds of properties of concurrent systems that designers are interested in. We mention three well-known classes of properties. For each of these classes, verification techniques exist that are based on an exhaustive traversal of the state space of a concurrent system. Hence, the state-space-reduction technique presented in this paper can be useful to improve these verification techniques. In the remainder, neither the details of the specification of properties nor the details of the verification techniques are very important. Therefore, we only give an informal explanation.

The first class of properties is the presence or absence of deadlocks. It is clear that deadlock properties can be verified in a straightforward way by means of an exhaustive state-space traversal.

The second class of properties are the so-called local properties. Local properties of a concurrent system are properties that typically depend only on the state of a single process of the system or on the state of a single shared object. The question whether or not a state satisfying some local property is reachable is essentially also verified by means of a state-space traversal. To verify whether a system state satisfies a local property, it is important to encode all relevant information concerning the property in the state labeling of the LTS representing the state space of the system. Thus, at this point, the reason for including a set of boolean propositions and an accompanying state labeling in the definition of an LTS becomes apparent. For more details on the verification of local properties, see [7, 11, 20].
The third class of properties are those expressible in (next-time-free) Linear-time Temporal Logic (LTL). Also LTL properties are formulated in terms of the propositions in an LTS. It is beyond the scope of this paper to give a formal definition of LTL; the interested reader is referred to [16]. The technique for verifying LTL formulae is referred to as (LTL) model checking. Again, the details are not important. For more information, see, for example, [12].

3 Partial-Order Reduction

Section 3.1 gives results, known from the literature on partial-order reduction (see, for example, [1, 7, 8, 11, 12, 18–22]), that are needed to prove that our reduction technique preserves deadlocks, local properties, and next-time-free LTL. Section 3.2 presents the partial-order-reduction algorithm of Holzmann and Peled [12, 19]. In Section 3.3, we briefly discuss implementation issues.

3.1 Basic theoretical framework

The basic idea of state-space-reduction techniques for enhancing verification is to restrict the part of the state space of a concurrent system that is explored during verification in such a way that properties of interest are preserved. There are several types of reduction techniques. In this paper, we focus on partial-order reduction. This technique exploits the independence of properties from the possible interleavings of the actions of the concurrent system. It uses the fact that the state-space explosion is often caused by the interleaving of independent actions of concurrently executing processes of the system (see Figure 1).

To be practically useful, a reduction of the state space of a concurrent system must be achieved during the traversal of the state space. This means that it must be decided per state which transitions, and hence which subsequent states, must be considered. Let $T = (S, \hat{s}, A, \tau, \Pi, L)$ be some LTS.

**Definition 2 (Reduction).** For any so-called reduction function $r : S \rightarrow 2^A$, we define the (partial-order) reduction of $T$ with respect to $r$ as the smallest LTS $T_r = (S_r, \hat{s}_r, A, \tau_r, \Pi, L_r)$ satisfying the following conditions:

- $S_r \subseteq S$, $\hat{s}_r = \hat{s}$, $\tau_r \subseteq \tau$, and $L_r = L \cap (S_r \times \Pi)$;
- for every $s \in S_r$ and $a \in r(s)$ such that $\tau_r(s, a)$ is defined, $\tau_r(s, a)$ is defined.

Note that these two requirements imply that, for every $s \in S_r$ and $a \in A$, if $\tau_r(s, a)$ is defined, then also $\tau(s, a)$ is defined and $\tau_r(s, a) = \tau(s, a)$.

It may be clear that not all reductions preserve all properties of interest. Thus, depending on the properties that a reduction must preserve, we have to define additional restrictions on $r$. To this end, we need to formally capture the notion of independence introduced earlier. Actions occurring in different processes may still influence each other, for example, when they access global variables. The following notion of independence defines the absence of such mutual influence.
Intuitively, two actions are independent iff, in every state where they are both enabled, (1) the execution of one action cannot disable the other and (2) the result of executing both actions is always the same.

**Definition 3 (Independence).** Actions \( a, b \in A \) with \( a \neq b \) are independent iff, for all states \( s \in S \) such that \( s \xrightarrow{a} \) and \( s \xrightarrow{b} \),

\[
- \tau(s, a) \xrightarrow{b} \text{ and } \tau(s, b) \xrightarrow{a}, \text{ and }
- \tau(\tau(s, a), b) = \tau(\tau(s, b), a).
\]

An example of independent actions are two assignments to or readings from local variables in distinct processes. Note that two actions are trivially independent if there is no state in which they are both enabled. It is straightforward to see that, in our running example of Figure 1, all actions are mutually independent.

The first class of properties we are interested in is the presence or absence of deadlocks. In order to preserve deadlock states of an LTS in a reduced LTS, the reduction function \( r \) must satisfy the following two conditions (called provisos):

- C0: \( r(s) = \emptyset \) iff \( \text{enabled}(s) = \emptyset \).
- C1 (persistency): For any \( s \in S \) and execution sequence \( s = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n \) of length \( n \in \mathbb{N} \setminus \{0\} \) such that, for all \( i \in \mathbb{N} \) with \( 0 \leq i < n \), \( a_i \not\in r(s) \), action \( a_{n-1} \) is independent of all actions in \( r(s) \).

The basic idea behind the persistency proviso is that, during the state-space traversal, transitions caused by actions that are independent of all the actions chosen by the reduction function can be ignored.

**Theorem 1 (Deadlock preservation [7, Theorem 4.3]).** Let \( r \) be a reduction function for LTS \( T \) that satisfies provisos C0 and C1. Any deadlock state reachable in \( T \) is also reachable in the reduced LTS \( T_r \) and vice versa.

A few remarks are in order. First, Theorem 4.3 in [7] does not state that any deadlock reachable in a reduced LTS is also reachable in the original LTS. However, this result follows immediately from proviso C0. Second, [7] uses a slightly stronger definition of independence. However, the proof of Theorem 4.3 in [7] carries over to our setting without change. Finally, observe that several authors presented state-space-reduction algorithms that preserve deadlocks [8, 18, 20].

Consider again Figure 1. It is not difficult to define a reduction function satisfying provisos C0 and C1 that reduces the LTS of Figure 1 to an LTS consisting of a single execution sequence from state 00 to state 28. Clearly, this reduction preserves deadlock state 28.

The second class of properties we discuss is the class of local properties. A local property is a boolean combination of propositions in \( \Pi \) whose truth value cannot be changed by two independent actions. That is, a property \( \phi \) is local iff, for all states \( s \in S \) and independent actions \( a, b \in A \) such that \( s \xrightarrow{a} \), \( s \xrightarrow{b} \), and \( \phi \) has different truth values in states \( s \) and \( \tau(s, a) \), the truth values of \( \phi \) in \( s \) and in \( \tau(s, b) \) are the same. An LTS satisfies a local property \( \phi \) iff there is a reachable
state that satisfies \( \phi \). Typical examples of local properties are properties that depend only on the state of a single process or shared object. To guarantee that a reduction of a state space preserves local properties, it suffices that the reduction function \( r \) satisfies the following requirement (in addition to C0 and C1).

- C2 (cycle proviso): For any cycle \( s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n = s_0 \) of length \( n \in \mathbb{N} \setminus \{0\} \), there is an \( i \in \mathbb{N} \) with \( 0 \leq i < n \) such that \( r(s_i) = enabled(s_i) \).

**Theorem 2 (Local-property preservation).** Let \( r \) be a reduction function for LTS \( T \) satisfying provisos C0, C1, and C2. Let \( \phi \) be a local property. LTS \( T \) satisfies \( \phi \) iff the reduced LTS \( T_r \) satisfies \( \phi \).

Proviso C2 prevents the so-called “ignoring problem” identified in [20]. Informally, the ignoring problem occurs when a reduction of a state space ignores the actions of an entire process. Proofs of (variants of) Theorem 2 can be found in \([7, 11, 20]\). In fact, these references show that proviso C2 can be weakened if one is only interested in the preservation of local properties. The stronger proviso given above is needed for the preservation of next-time-free LTL properties.

The third class of properties we are interested in is the class of properties expressible in next-time-free LTL. For any LTL formula \( \phi \), \( prop(\phi) \) is the set of propositions occurring in \( \phi \).

**Definition 4 (Invisibility).** An action \( a \in A \) is \( \phi \)-invisible in state \( s \in S \) iff \( \tau(s, a) \) is undefined or, for all \( \pi \in prop(\phi), \pi \in L(s) \iff \pi \in L(\tau(s, a)) \). Action \( a \) is globally \( \phi \)-invisible iff it is \( \phi \)-invisible for all \( s \in S \).

Informally, an action is globally \( \phi \)-invisible iff it cannot change the truth value of formula \( \phi \).

Given a next-time-free LTL formula \( \phi \) and assuming that a reduction function \( r \) satisfies the following proviso in addition to C0, C1, and C2, it can be shown that the reduced LTS satisfies \( \phi \) iff the original LTS satisfies \( \phi \). For a precise definition of the satisfaction of an LTL formula by some LTS, see [16].

- C3 (invisibility): For any state \( s \in S \), all actions in \( r(s) \) are globally \( \phi \)-invisible or \( r(s) = enabled(s) \).

**Theorem 3 (Next-time-free LTL preservation [19, 21]).** Let \( r \) be a reduction function for LTS \( T \) satisfying C0, C1, C2, and C3. Let \( \phi \) be a next-time-free LTL formula. LTS \( T \) satisfies \( \phi \) iff the reduced LTS \( T_r \) satisfies \( \phi \).

### 3.2 The partial-order-reduction algorithm of Holzmann and Peled

Given the three theorems of the previous subsection, the challenge is to find interesting reduction functions and efficient algorithms implementing the corresponding reductions. A well-known partial-order-reduction algorithm is the one described in \([12, 19]\). The most important aspects of the algorithm are the following: (1) It is based on a depth-first search (DFS) of the state space of a concurrent
system and (2) it uses a reduction function based on the process structure of the system. For the full details of the algorithm, the reader is referred to the original references [12, 19]. In this paper, we concentrate on the reduction function. To this end, we introduce a notion of processes in our framework of LTSs.

Let \( T = (S, s, A, \tau, I, L) \) be some LTS. We assume that a given set of processes \( P \) is associated with \( T \) as follows. Each process \( P \in P \) is a set of actions, i.e., \( P \subseteq A \). We require that \( A = \bigcup_{P \in P} P \) and that, for all \( P, Q \in P \) with \( P \neq Q \), \( P \cap Q = \emptyset \). That is, the set of processes partitions the set of actions. Function \( Pid : A \to P \) yields for each action the process it is contained in. Not every partitioning of actions yields a meaningful process structure. Concurrency within processes is not allowed. Therefore, we require the following. Let \( a, b \in A \) with \( a \neq b \) be a pair of distinct actions belonging to the same process in \( P \). For any state \( s \in S \) such that \( a, b \in enabled(s) \), \( b \notin enabled(\tau(s, a)) \). That is, each two actions from a single process that are simultaneously enabled in a given state must disable each other. Clearly, this restriction disallows concurrency within processes, whereas it does allow choices.

The following definition is crucial in the formulation of the abovementioned reduction function. It depends on the class of properties one is interested in.

**Definition 5 (safety).** An action \( a \in A \) is said to be safe iff it is independent from any (other) action \( b \in A \) with \( Pid(b) \neq Pid(a) \). An action \( a \in A \) is said to be safe for a given next-time-free LTL formula \( \phi \) iff it is independent from any action \( b \in A \) with \( Pid(b) \neq Pid(a) \) and globally \( \phi \)-invisible.

As mentioned, the algorithm of [12, 19] performs the reduction of the state space during a DFS. A DFS uses a stack to store partially investigated states. The reduction is obtained by defining for each state a so-called ample set. Note that the definition uses safety and, hence, depends on the particular class of properties to be verified.

**Definition 6 (Reduction function ample).** Let \( s \in S \). Consider the set \( SP \) of processes \( P \in P \) such that:
- \( enabled(s) \cap P \neq \emptyset \),
- for all \( a \in enabled(s) \cap P \), \( a \) is safe (for some next-time-free LTL formula \( \phi \)),
- and \( \tau(s, a) \) is not on the DFS stack.

If \( SP \) is empty, then define \( ample(s) = enabled(s) \); otherwise, choose an arbitrary element \( P \) of \( SP \) and define \( ample(s) = enabled(s) \cap P \). Set \( ample(s) \) is said to be the ample set for \( s \).

It is not difficult to verify that reduction function \( ample \) satisfies provisos C0 through C3 given in the previous subsection. C0 follows easily from Definition 6. C1 and C3 follow from the safety requirement in Definition 6. Finally, C2 follows from the requirement in Definition 6 that the state resulting from the execution of an action in the ample set cannot be on the DFS stack unless the ample set consists of the entire set of enabled actions. As a result, the reduction via \( ample \) preserves deadlocks (Theorem 1), local properties (Theorem 2), and next-time-free LTL (Theorem 3). For more details, see [12, 19].
3.3 Implementation in SPIN

SPIN [9] is a tool supporting the automatic verification of deadlock, local, and next-time-free LTL properties. Specifications of concurrent system are written in the language PROMELA. The partial-order-reduction algorithm of [12, 19] has been implemented in SPIN. To allow for the efficient computation of ample sets during a DFS, sufficient conditions for the safety of actions are derived from the PROMELA specification before starting the DFS (see [12]). For instance, a sufficient condition for the safety of an action that can be derived from a PROMELA specification is that it does not touch any global objects such as variables or communication channels. Another good reference for readers interested in implementation issues concerning partial-order reduction is [7].

4 Cluster-based Partial-Order Reduction

We motivate our improvement of partial-order reduction by means of two different specifications of the same concurrent system given in Figure 2. Specification example1 has two global variables and four processes, each of them executing a single action assigning a value to one of these variables. Clearly, none of the actions is independent of all the other actions, which by Definition 5 means that none of the actions is safe. Thus, reduction function ample of Definition 6 yields no reduction. The interested reader could verify that the LTS corresponding to the concurrent system has 25 states and 40 transitions.

![Fig. 2. Two specifications of a concurrent system and a reduced state space.](image)

Specification example2 in Figure 2 is a variant of example1 with the processes clustered in pairs. The two clusters encapsulate the dependencies between processes. As a result, all actions within one cluster are independent of all actions within the other one. Our idea is to augment an LTS with a hierarchy of clusters and to generalize Definitions 5 and 6 to clusters. Thus, it is possible to reduce the state space of the system of Figure 2 to an LTS that includes all interleavings of actions within clusters $C_0$ and $C_1$ but only a single interleaving of actions from different clusters (see Figure 2), while preserving all properties of interest. Let $T = (S, s, A, \tau, \Pi, L)$ be an LTS with processes $P$. 
Definition 7 (Clustering). A cluster of processes in \( \mathcal{P} \) is simply a set of processes. A clustering \( \mathcal{C} \subseteq 2^\mathcal{P} \) of \( \mathcal{P} \) is a set of clusters partitioning \( \mathcal{P} \), i.e., \( \mathcal{P} = \bigcup_{\mathcal{C} \in \mathcal{C}} \mathcal{C} \) and, for all \( \mathcal{C}, \mathcal{D} \in \mathcal{C} \) with \( \mathcal{C} \neq \mathcal{D} \), \( \mathcal{C} \cap \mathcal{D} = \emptyset \). A clustering hierarchy \( \mathcal{H} \) for \( \mathcal{P} \) is a finite ordered set \( \{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_{n-1}\} \), where \( n \in \mathbb{N} \setminus \{0\} \), of clusterings of \( \mathcal{P} \) such that, for all \( i \in \mathbb{N} \) with \( 0 < i < n \),

- \( |\mathcal{C}_i| < |\mathcal{C}_{i-1}| \) and
- for each \( \mathcal{C} \in \mathcal{C}_{i-1} \), there exists a \( \mathcal{D} \in \mathcal{C}_i \) such that \( \mathcal{C} \subseteq \mathcal{D} \).

For any \( i \in \mathbb{N} \) with \( 0 \leq i < n \), clustering \( \mathcal{C}_i \) is called level \( i \) of the hierarchy. Level \( \mathcal{C}_i \) is above level \( \mathcal{C}_j \) iff \( i > j \); it is below \( \mathcal{C}_j \) iff \( i < j \).

Clearly, Definition 7 implies that each level in a clustering hierarchy is a coarsening of all lower levels. Also note that the maximum number of levels in a clustering hierarchy is limited by the number of processes in \( \mathcal{P} \). A clustering hierarchy for \text{example2} consisting of three levels (numbered 0, 1, and 2) is the following: \( \{\{\{P_0\}, \{P_1\}, \{P_2\}, \{P_3\}\}, \{C_0 = \{P_0, P_1\}, C_1 = \{P_2, P_3\}\}, \{\{P_0, P_1, P_2, P_3\}\}\} \).

Let \( \mathcal{H} = \{\mathcal{C}_0, \ldots, \mathcal{C}_{n-1}\} \), with \( n \in \mathbb{N} \setminus \{0\} \), be a clustering hierarchy for \( \mathcal{P} \). For all \( i \in \mathbb{N} \) with \( 0 \leq i < n \), function \( \text{Cid}^{(i)}: \mathcal{P} \rightarrow \mathcal{C}_i \) yields for a given process the level-\( i \) cluster it belongs to; that is, given a process \( \mathcal{P} \in \mathcal{P} \) and level \( i \in \mathbb{N} \) with \( 0 \leq i < n \), \( \text{Cid}^{(i)}(\mathcal{P}) = \mathcal{C} \) with \( \mathcal{C} \in \mathcal{C}_i \) the unique cluster such that \( \mathcal{P} \in \mathcal{C} \).

Definition 8 (Level-\( i \) safety). Let \( i \in \mathbb{N} \) with \( 0 \leq i < n \). Action \( a \in A \) is level-\( i \) safe (for a given next-time-free LTL formula \( \phi \)) iff it is independent of any action \( b \in A \) with \( \text{Cid}^{(i)}(\text{Pid}(b)) \neq \text{Cid}^{(i)}(\text{Pid}(a)) \) (and globally \( \phi \)-invisible).

In the above clustering hierarchy for \text{example2}, all actions are level-1 and (trivially) level-2 safe, but not level-0 safe.

Given a cluster \( \mathcal{C} \subseteq \mathcal{P} \), let \( \text{actions}(\mathcal{C}) = \bigcup_{\mathcal{P} \in \mathcal{P}} \mathcal{P} \). Furthermore, for any state \( s \in S \), let \( \text{enabled}(s, \mathcal{C}) = \text{actions}(\mathcal{C}) \cap \text{enabled}(s) \).

Definition 9 (Reduction function \text{ample}). Let \( s \in S \). Let, for each level \( \mathcal{C}_i \in \mathcal{H} \), \( \mathcal{S}_{\mathcal{C}_i} \) be the set of clusters \( \mathcal{C} \in \mathcal{C}_i \) such that \( \text{enabled}(s, \mathcal{C}) \neq \emptyset \), for all \( a \in \text{enabled}(s, \mathcal{C}) \), \( a \) is level-\( i \) safe (for some next-time-free LTL formula \( \phi \)), and \( \tau(s, a) \) is not on the DFS stack. If the set \( \mathcal{S}_{\mathcal{C}_i} \) is empty for all levels \( \mathcal{C}_i \), then define \( \text{ample}(s) = \text{enabled}(s) \); otherwise, choose a level \( \mathcal{C}_i \) such that the set \( \mathcal{S}_{\mathcal{C}_i} \) is non-empty, select an arbitrary element \( \mathcal{C} \in \mathcal{S}_{\mathcal{C}_i} \), and define \( \text{ample}(s) = \text{enabled}(s, \mathcal{C}) \).

It is interesting to observe that function \text{ample} of Definition 6 is a special case of function \text{ample} of Definition 9 with a trivial hierarchy of only one level that consists of trivial clusters each containing only one process.

Theorem 4. The reduction of an LTS obtained via reduction function \text{ample} of Definition 9 preserves deadlock-, local-, and next-time-free LTL properties.

Proof. It is straightforward to prove that \text{ample} satisfies provisos C0, C1, C2, and C3 of Section 3.1. The arguments are the same as in Section 3.2, where it is argued that function \text{ample} of Definition 6 satisfies these provisos. The desired result follows immediately from Theorems 1, 2, and 3.
Consider again the concurrent system of Figure 2. The figure shows a reduced state space of this system. In each state, the values of variables $u$ and $v$ are given with $\perp$ meaning that a value is undefined. The sets of actions chosen in each state are ample sets as defined by function $\text{ample}$ of Definition 9. The reduced state space has 13 states and 12 transitions. This means reductions of the complete state space, consisting of 25 states and 40 transitions, of 48% in states and 70% in transitions. (Recall that standard partial-order reduction yields no reductions.)

In practice, the largest state-space reductions are obtained by choosing the ample sets as small as possible. A simple way to obtain small ample sets is to search the levels in a clustering hierarchy for suitable clusters in increasing order.

Another practical issue is how to obtain useful clustering hierarchies. Such a clustering should maximize the dependencies between processes within a cluster and minimize the dependencies between clusters. It is our aim to obtain such hierarchies statically, derived from the concurrent-system specification. In that way, we avoid the overhead of forming a hierarchy on-the-fly by inspecting dependencies between processes during the DFS. One possibility to derive a hierarchy in a static way is to preprocess the system specification and to cluster processes based on shared objects. Another option is to use the existing hierarchical or modular structure of a specification; many contemporary specification and modeling languages such as UML and SDL include standard hierarchical structuring mechanisms. A final option could be the use of advanced statistical clustering techniques (based on run-time information) as described in [14].

5 Experiments

To validate our cluster-based reduction algorithm, we implemented a prototype on top of the verification tool SPIN, version 3.2.4. We applied our prototype implementation to several examples. Since PROMELA, the input language of SPIN, does not support modular design, we provided the clustering hierarchies ourselves (based on a straightforward informal analysis of the examples). Our focus is on the generation of state spaces; we did not verify any properties. The goal of the experiments is to compare reductions in states and transitions obtained via our algorithm with reductions obtained via SPIN’s standard partial-order reduction. As may be expected, similar to standard partial-order reduction, our algorithm performs best for systems with a large amount of concurrency. In all cases, we observe significantly larger reductions than the ones obtained with standard partial-order reduction. Moreover, in all cases, our algorithm reduces verification times. The overhead of upgrading the standard partial-order-reduction engine is marginalized by the gain in time because of the smaller number of generated states and transitions.

In the remainder, we show the results of three case studies. The experiments were performed on a Sun Ultra-10 machine, with a 299 MHz UltraSPARC-IIi processor and 128 MB of main memory, running the SunOS 5.6 operating system. All the verification times are given in seconds.
**Best-case example.** Our first case study consists of variants of system example2 of Figure 2. The systems we verified consist of N pairs of processes and N variables, each pair sharing one of these variables. The system with N=2 corresponds to system example2. Table 1 shows the obtained results, including the reductions in states and transitions in percentages compared to SPIN with standard partial-order reduction. The numbers deviate from the theoretical results given in Section 4 due to implementation details of SPIN. SPIN adds to each process a special end transition that is independent of all other transitions. The standard SPIN partial-order reduction captures the independence of these special end transitions. Our prototype implementation takes, in addition, advantage of the independence of some transitions involving shared variables, as explained in Section 4. We used a hierarchy of two layers, with the nontrivial layer consisting of clusters that coincided with process pairs.

<table>
<thead>
<tr>
<th>without POR</th>
<th>standard POR</th>
<th>cluster-based POR</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>states</td>
<td>trans</td>
</tr>
<tr>
<td>2</td>
<td>65</td>
<td>108</td>
</tr>
<tr>
<td>3</td>
<td>329</td>
<td>784</td>
</tr>
<tr>
<td>4</td>
<td>1657</td>
<td>5216</td>
</tr>
<tr>
<td>5</td>
<td>8313</td>
<td>32680</td>
</tr>
</tbody>
</table>

**Table 1.** Results for the best-case example.

**Parity Computer.** The second example, taken from [2], models a Parity Computer with a tree structure. The system consists of a root module, N client modules as leaves, and join modules as intermediate nodes. The system with N=4 is shown in Figure 3. The figure also shows a clustering hierarchy that is immediately derived from the modular structure of the Parity Computer.
A client process starts with nondeterministically generating a bit value that it puts into the request variable that it shares with its parent join module. Subsequently, it continuously waits for an acknowledgment from its parent. Each time it receives an acknowledgment, it again sends an arbitrary bit value to its parent. A join process computes the parity (XOR) of its two inputs and transmits it upwards to its parent, while, simultaneously, sending an acknowledgment to its children. Eventually, parity bits are delivered to the root process.

Consider again the example in Figure 3. Our cluster-based partial-order-reduction algorithm takes advantage of the fact that a join process communicates with its parent and its children in alternating order. Because the cluster with root Join0 is independent of the cluster with root Join1, we can reduce the state space by basically serializing the transitions internal to one of these clusters with those internal to the other one. SPIN’s standard reduction algorithm does not give any reduction because all transitions involve global variables. As Table 2 shows, the reduction with cluster-based partial-order reduction is quite impressive.

<table>
<thead>
<tr>
<th></th>
<th>standard POR</th>
<th>cluster-based POR</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>states</td>
<td>trans time</td>
</tr>
<tr>
<td>4</td>
<td>1749</td>
<td>4798</td>
</tr>
<tr>
<td>5</td>
<td>7933</td>
<td>27012</td>
</tr>
<tr>
<td>6</td>
<td>69615</td>
<td>288678</td>
</tr>
<tr>
<td>7</td>
<td>320095</td>
<td>1213520</td>
</tr>
<tr>
<td>8</td>
<td>2782640</td>
<td>15581300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>states</th>
<th>trans time</th>
<th>states</th>
<th>% trans</th>
<th>% time</th>
</tr>
</thead>
</table>
| a        | Because of the large memory requirements, this experiment was performed on a Sun Ultra-Enterprise machine with three 248 MHz UltraSPARC-II processors and 2304 MB of main memory, running the SunOS 5.5.1 operating system. Table 2. Results for the Parity Computer.

The reduction with cluster-based partial-order reduction is slightly worse (though of the same order of magnitude) than the reduction reported in [2] for the same examples, obtained with the Next heuristic. However, it is difficult to draw any final conclusions about the comparative efficiency of the two techniques based only on this one example. First, the input languages in which the models are specified are different, which inevitably leads to differences in the model. Second, the Parity-Computer example is one of the best cases for the Next heuristic and, as the authors note themselves in [2], there are many examples for which partial-order reduction gives better results than the Next heuristic.

**Concurrent Alternating-Bit Protocol.** Finally, we consider the Concurrent Alternating-Bit Protocol (CABP) of [13]. The CABP consists of six components, as depicted in Figure 4. Each component is modeled as a separate process.

The CABP uses the standard alternating-bit scheme to avoid communication errors. Component A fetches data from its environment and transmits this data repeatedly through channel $K$ until an acknowledgment is received from $D$. 
A does not wait for a negative response before retransmitting. Channel \( K \) is unreliable in the sense that it can corrupt or lose data. The role of \( B \) is to forward successfully received data to the environment; each correct reception is acknowledged to \( C \). \( C \) transmits acknowledgments repeatedly via unreliable channel \( L \). \( D \) receives acknowledgments from \( L \) and passes them to \( A \).

As for the Parity Computer, also for the CABP the standard reduction algorithm does not produce any reduction; all transitions involve communications through global synchronous (rendez-vous) channels. The cluster-based reduction algorithm, however, capitalizes on part of the concurrency between the system modules, as the results in Figure 4 show. The hierarchy consists of two levels, with nontrivial level \( \{ A \} \), \( \{ K, B \} \), \( \{ C \} \), and \( \{ L, D \} \). The implementation of \( A \) and \( C \) is such that each action in these processes depends on both its neighbors. Thus, any other process clustering does not improve the results.

### 6 Conclusion

The main contribution of this paper is an enhancement of the partial-order-reduction scheme of [12, 19]. Using the inherent structure of concurrent systems, we improve the way the safety (i.e., independence and invisibility) of actions is determined syntactically during the compilation of the system specification. The resulting ample sets may contain actions from more than one process. Although ample sets with actions from several processes have been considered earlier (e.g., [1]), to the best of our knowledge, the idea of exploiting hierarchical system structure is not present in the literature. We implemented our algorithm on top of SPIN, by upgrading SPIN’s standard partial-order-reduction engine. The prototype implementation has been tested on several examples known from the literature and the obtained results are encouraging: Compared to SPIN’s standard partial-order-reduction algorithm, significantly larger reductions of state spaces are obtained and verification times are decreased.

It will be interesting to see how our approach works in combination with other state-space-reduction heuristics. Following [6], it is easy to show that our technique is fully compatible with symmetry reduction. The two techniques are orthogonal because they exploit different features of concurrent systems. We agree with the conjecture in [2] that partial-order reduction (and also our enhancement) is compatible with the Next heuristic. It seems though that cluster-based partial-order reduction and the Next heuristic are not fully orthogonal, because they capture to some extent the same redundancies in state spaces. It
is interesting to study the relation between the Next heuristic and cluster-based partial-order reduction in more detail.

The main task in the near future is to fully automate our implementation. In the current prototype, the clustering hierarchy and safety levels must be included manually. A fully automatic implementation will allow us to test the implementation on larger, real-world examples. We also plan to take advantage of the improved partial-order reduction [10] introduced in the latest releases of SPIN, as well as the introduction of V-Promela [15]. Another interesting topic is the study of clustering heuristics (see, e.g., [14]). Good clustering heuristics might actually yield better clustering hierarchies than the ones obtained from the hierarchical structure specified by a system designer. Finally, our cluster-based algorithm is compatible with the upgrade of SPIN’s engine for timed systems from [4]. It is very likely that it can be combined with recent attempts of partial-order reduction for timed automata [3, 5, 17].

References


3. Partial-Order Reduction in Presence of Rendez-vous Communications with Priority Choice and Weak Fairness

This chapter is a revised version of:

D. Bošnački, Partial-Order Reduction in Presence of Rendez-vous Communications with Unless Constructs and Weak Fairness, Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Lecture Notes in Computer Science 1680, Springer-Verlag, 1999, pp. 40–56
Partial Order Reduction in Presence of Rendez-vous Communications with Priority Choice and Weak Fairness

Dragan Bošnački *

Dept. of Computing Science, Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
fax: +31 40 246 3992, e-mail: dragan@win.tue.nl

Abstract. If synchronizing (rendez-vous) communications are used in Promela models, the priority choice feature (unless construct) and the weak fairness algorithm are not compatible with the partial order reduction algorithm used in Spin’s verifier. After identifying the wrong partial order reduction pattern that causes both incompatibilities, we give solutions to these two problems. To this end we propose corrections in the identification of the safe statements for partial order reduction and, as an alternative, we discuss corrections of the partial order reduction algorithm.

1 Introduction

The issue of fairness is an inherent and important one in the study of concurrency and nondeterminism, in particular in the area of verification of concurrent systems. Since fairness is used as generic notion there is a broad taxonomy of fairness concepts. In this chapter we confine our attention to the notion of weak fairness on the level of processes which is implemented in the Spin verifier. This means that we require that for every execution sequence of the concurrent program which is a composition of several processes, if some process becomes continuously enabled at some point of time (i.e. can always execute some of its statements), then at least one statement from that process will eventually be executed. This kind of fairness is most often associated with mutual exclusion algorithms, busy waiting, simple queue-implementations of scheduling, and resource allocation. Weak fairness will guarantee the correctness of statements like eventually entering the critical region for every process which is continuously trying to do that (in the mutual exclusions) or eventually leaving the waiting queue for each process that has entered it (in the scheduling) [10].

Partial order reduction is one of the main techniques that are used to alleviate the problem of state space explosion in the verification of concurrent systems.

* On leave from the Institute of Informatics, Faculty of Natural Sciences and Mathematics, University “Sts. Cyril and Methodius”, Skopje, Macedonia. Supported by EC ESPRIT LTR Project No. 23498 (VIRES).
and it is indeed one of Spin’s main strengths. The idea is, instead of exploring all the execution sequences of a given program, to group them in equivalence classes which consist of interleavings of independent program statements. Then only representatives for each equivalence class are considered. In practice this is realized such that from each state only a subset of the executable statements are taken.

Combining the algorithms for model-checking under weak fairness with partial order reduction is a prerequisite for the verification of many interesting properties to be feasible in practice. However, it was discovered [7] that the two algorithms are not compatible when rendez-vous communications occur in the Promela models. As a result, in the present implementation of Spin (version 3.3 and later) the combination of weak fairness with partial order reduction when rendez-vous are used in the models is not allowed.

Another problem with Spin’s partial order reduction in presence of rendez-vous occurs when the unless construct is used in the Promela models. The combination of these three Spin features is also currently forbidden. As unless is a natural way to represent exceptions, and the latter gains more and more popularity in modern programming languages (e.g. Java), this incompatibility can be a serious drawback.

Interestingly, it turns out that both incompatibilities are caused by exactly the same pattern of wrong partial order reduction. After pointing out this incorrect reduction pattern we propose solutions to the problems with fairness and unless. For both cases we discuss two kinds of solutions, classified according to the two different phases of the verification in which they are implemented. The first kind corrects the identifications of so called safe statements for the partial order reduction algorithm. The marking of statements as safe is done during the compilation of the Promela model, so we call these solutions static. The second kind are the dynamic solutions which are applied during the exploration of the state space and are in fact corrections of the partial order reduction algorithm.

In the next section we give the necessary preliminaries for the rest of the chapter. Section 3 is devoted to partial order reduction and the concrete algorithm that is used in Spin. In Section 4 we discuss the problem with the unless construct and give some solutions to overcome it. Section 5 deals with the Spin’s weak fairness algorithm. We first present the algorithm and show its correctness. After locating the problem and comparing it to the priority choice (unless) case, we again propose both static and dynamic solutions. The last section is a summary with some considerations concerning future work.

2 Preliminaries

In this section, following [14] and [6], we give the semantics of Promela programs (models) and their verification in terms of finite labeled transition systems.

We represent the programs as collections of processes. The semantics of the process $P_i$ can be represented as a labeled transition system (LTS). An LTS
is a quadruple \((S_i, \dot{s}_i, \tau_i, L_i)\), where \(S_i\) is a finite set of states, \(\dot{s}_i \in S_i\) is a distinguished initial state, \(L_i\) is a finite set of program statements (labels), and \(\tau_i : S_i \times L_i \rightarrow 2^{S_i}\) is a transition function. The transition function induces the set \(R_i \subseteq S_i \times L_i \times S_i\) of transitions. Every transition in \(R_i\) is the result of an execution of a statement from the process, and \((s_i, a, s'_i) \in R_i\) iff \(s'_i \in \tau_i(s_i, a)\).

We introduce a function \(\text{Label}\) that maps each transition to the corresponding statement. For a statement \(a\), with \(\text{Pid}(a)\) we denote the process to which \(a\) belongs. (If two syntactically identical statements belong to different processes we consider them as different.) A statement \(a\) is enabled in some state \(s \in S_i\) iff there exists \(s' \in S_i\) such that \(s' \in \tau(s, a)\). In this case we also say that the transition \(s \xrightarrow{a} s'\) is enabled in \(s\). (The enabledness (executability) of a given statement is obtained from the process specification according to rules that we do not consider here.) \(\text{En}(a)\) denotes the set of states in which \(a\) is enabled.

Given a state \(s\) and process \(p\) we say that \(p\) is enabled in \(s\) (we write \(s \in \text{En}(p)\)) if there is a statement \(a\) such that \(\text{Pid}(a) = p\) and \(s \in \text{En}(a)\).

Now we can define the semantics of the program \(P\) that corresponds to the concurrent execution of the processes \(P_i (1 \leq i \leq N)\) as an LTS which is a product of the labeled transition systems corresponding to the component processes. In the definition we pay a special attention to the transitions generated by rendez-vous communications. We model each such communication as an atomic sequence of a rendez-vous send statement \(a\) and a corresponding rendez-vous receive statement \(a'\). The statements \(a\) and \(a'\) generate the consecutive transitions \(t\) and \(t'\), respectively, such that \(\text{Label}(t) = a\) and \(\text{Label}(t') = a'\). To preserve the atomicity of the execution, it is required that \(t'\) is the only outgoing transition from the intermediate global state obtained after the execution of \(t\). The product LTS \((S, \dot{s}, \tau, L)\) consists of:

- state space \(S = \prod_{1 \leq i \leq N} S_i\), i.e., the Cartesian product of the state spaces \(S_i\),
- initial state \(\dot{s} = (\dot{s}_1, \ldots, \dot{s}_N)\),
- \(L = \bigcup_{1 \leq i \leq N} L_i\), i.e., the set of statements is the union of the statement sets of the components,
- and the transition function defined as:
  1. if \(a\) is not a rendez-vous statement, then \((s_1, \ldots, s'_k, \ldots, s_N, a) \in \tau(s_1, \ldots, s_k, \ldots, s_N, a)\) iff \(s'_k \in \tau_k(s_k, a)\)
  2. if \(a\) is a rendez-vous send, and there is a rendez-vous receive statement \(a'\), such that \(\text{Pid}(a') \neq \text{Pid}(a)\), \(s'_k \in \tau_k(s_k, a)\) and \(s'_l \in \tau_l(s_l, a')\), then
     (a) \((s_1, \ldots, s'_k, \ldots, s_l, \ldots, s_N, a) \in \tau(s_1, \ldots, s_k, \ldots, s_l, \ldots, s_N, a)\)
     (b) \((s_1, \ldots, s'_k, \ldots, s'_l, \ldots, s_N) \in \tau(s_1, \ldots, s_k, \ldots, s'_l, \ldots, s_N, a')\)
     (c) There is no action \(a'' \neq a'\) such that \((s_1, \ldots, s'_k, \ldots, s_l, \ldots, s_N, a) \in \text{En}(a'')\), i.e., no other statement can be interleaved with the execution of receive;

A popular way to represent requirements on the program is by a linear temporal logic (LTL) formula \([9]\). In Spin, next-time-free LTL is used, which means that formulae may contain only boolean propositions on system states, the boolean
operators \( \land, \lor, ! \) (negation), and the temporal operators \( \Box \) (always), \( \Diamond \) (eventually) and \( U \) (until). For verification purposes the LTL formulae are translated into Büchi automata.

A Büchi automaton is a tuple \( B = (\Sigma, S, \rho, \hat{s}, F) \), where:

- \( \Sigma \) is an alphabet,
- \( S \) is a set of states,
- \( \rho : S \times \Sigma \to 2^S \) is a transition function,
- \( \hat{s} \in S \) is the initial state,
- and \( F \subseteq S \) is a set of designated states called acceptance states.

A run of \( B \) over an infinite word \( w = a_1a_2 \ldots \), is an infinite sequence \( s_0s_1 \ldots \) of states, where \( s_0 \) is the initial state \( \hat{s} \) and \( s_i \in \rho(s_{i-1}, a_i) \), for all \( i \geq 1 \). A run \( s_0s_1 \ldots \) is accepting if for some \( s \in F \) there are infinitely many \( i \)'s such that \( s_i = s \). The word \( w \) is accepted by \( B \) if there is an accepting run of \( B \) over \( w \).

The transitions of the Büchi automaton that is obtained from the formula are labeled with boolean propositions over the global system states of the LTS corresponding to the program, i.e., the alphabet \( \Sigma \) is a set of boolean propositions.

In order to prove the satisfaction of the LTL formula by the program \( P \), we further define the synchronous product of the LTS \((S_P, \hat{s}_P, \tau_P, L_P)\) corresponding to \( P \) and the Büchi automaton \( A = (\Sigma, S_A, \rho, \hat{s}_A, F_A) \) obtained from the negation of the LTL formula, to be an LTS extended with acceptance states, i.e., extended LTS (XLTS) \((S, \hat{s}, \tau, L, F)_1\) with:

- state set \( S = S_P \times S_A \),
- initial state \( \hat{s} = (\hat{s}_P, \hat{s}_A) \),
- transition function \( \tau : S \times L \to 2^S \) defined as \((s_{2P}, s_{2A}) \in \tau(s_{1P}, s_{1A}, a) \) iff \( s_{2P} \in \tau_P(s_{1P}, a) \) and there is a proposition \( p \in \Sigma \) such that \( s_{2A} \in \rho(s_{1A}, p) \) and \( p \) is true in \( s_{1P} \),
- set of statements \( L = L_P \);
- and a set of designated acceptance states \( F = \{(s_P, s_A) \mid s_A \in F_A\} \), i.e. we declare as acceptance states the states with second component belonging to the acceptance set of the Büchi automaton \( A \).

Unless stated differently, for the rest of the chapter we fix the XLTS \( T \) to be the tuple \((S, \hat{s}, \tau, L, F)_1\) as defined above. With \( R \) we denote the set of transitions of \( T \).

An execution sequence or path is a finite or infinite sequence of subsequent transitions, i.e., for \( s_i \in S, a_i \in L \), the sequence \( s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots \) is an execution sequence in \( T \) iff \( s_i \xrightarrow{a_i} s_{i+1} \in R \) for all \( i \geq 0 \). An infinite execution sequence is said to be accepting if it starts in the initial state \( \hat{s} \) and there is an acceptance

\[1\] Although the proliferation of different formal models (LTS, Büchi automata, extended LTS) that are used to represent the semantics and the state space might seem unnecessary, we use three different formal concepts in order to follow more closely [14] and [6], so that we will be able to reuse most of the results from these papers in a seamless way.
state $s \in F$ that occurs infinitely many times in the sequence. A finite execution sequence $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n (n \geq 1)$ is a cycle iff the start and end states coincide, i.e. $s_0 = s_n$. Given a finite or infinite execution sequence $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$, a process $P_i, 1 \leq i \leq N$, and a state $s_j$ from the execution sequence, we say that $P_i$ is executed in $\sigma$ in $s_j$ iff $\text{Pid}(a_j) = i$. A state $s$ is reachable iff there exists a finite execution sequence that starts at $s$ and ends in $s$. A cycle $c$ is reachable iff there exists a state in $c$ which is reachable. A cycle $c$ is an acceptance cycle if it contains at least one acceptance state.

The satisfaction of the formula by the program $P$ can now be proven by showing that there are no accepting execution sequences of the extended LTS $T$. On the other hand, the existence of accepting execution sequences means that the formula is not satisfied. From the definition of Büchi automata and extended LTS and following the reasoning from [6], for instance, it is straightforward to conclude that the extended LTS has an accepting execution sequence if it has some state $s \in F$ that is reachable from the initial state and reachable from itself (in one or more steps) [6]. Thus, we have to look for reachable acceptance cycles in the XLTS $T$. In the sequel we call an XLTS also a state space.

### 3 Partial Order Reduction

In this section we give a brief overview of the partial order reduction (POR) algorithm by Holzmann and Peled [14], that is considered throughout the chapter. This algorithm is also implemented in Spin. We start by rephrasing some definitions from [14].

The basic idea of the reduction is to restrict the part of the state space that is explored by the DFS, in such a way that the properties of interest are preserved. To this purpose, the independence of the checked property from the possible interleaving of statements is exploited. More specifically, two statements $a, b$ are allowed to be permuted precisely then, if for all sequences $v, w$ of statements: if $vabw$ (where juxtaposition denotes concatenation) is an accepted behavior, then $vbaw$ is an accepted behavior as well. In practice, sufficient conditions for such permutability are used that can be checked locally, i.e., in a state. For this, a notion of “concurrency” of statements is used that captures the idea that transitions are contributed by different, concurrently executing processes of the system.

We first introduce some additional terminology. Without loss of generality we assume that the transition function $\tau$ of the LTS representing the program is deterministic, i.e., the set $\tau(s,a)$ consists of at most one element, for any state $s \in S$ and any statement $a \in L$. For $q \in En(a)$, let $a(q)$ be the state which is reached by executing $a$ in state $q$. Concurrent statements (i.e. statements with different Pids) may still influence each other’s enabledness, whence it may not be correct to only consider one particular order of execution from some state. The following notion of independence defines the absence of such mutual influence. Intuitively, two statements are independent if in every state where they are both
enabled, they cannot disable each other, and are commutative, i.e., the order of their execution makes no difference to the resulting state.

**Definition 1.** The statements $a$ and $b$ are independent iff for all states $q$ such that $q \in \text{En}(a)$ and $q \in \text{En}(b)$,

\begin{itemize}
  \item $a(q) \in \text{En}(b)$ and $b(q) \in \text{En}(a)$, and
  \item $a(b(q)) = b(a(q))$.
\end{itemize}

*Statements that are not independent are called dependent.*

Note that $a$ and $b$ are trivially independent if $\text{En}(a) \cap \text{En}(b) = \emptyset$. An example of independent statements are assignments to or readings from local variables, executed by two distinct processes.

Also note that the statements $a$ and $b$ are considered to be independent even if $a$ can enable $b$ (and vice versa). The main requirement is that the statements do not disable each other. This is unusual in a sense, because in the literature a more strict definition prevails that does not allow that a statement can enable another statement (e.g. [20, 11]). The advantage of the subtlety in Definition 1 is that it ensures a greater set of independent statements than the “classical” definition and consequently a better reduction of the state space. However, we must be careful with this, because as we will see later, this feature is closely connected with the incompatibilities that we are discussing in this chapter.

Another reason why it may not be correct to only consider only one particular order of execution from state $s$ of two concurrent statements $a$ and $b$ is that the difference between the intermediate states $a(s)$ and $b(s)$ may be observable in the sense that it influences the property to be checked. For a given proposition $p$ that occurs in the property (an LTL formula), and a state $s$, let $p(s)$ denote the boolean value of the proposition $p$ in the state $s$. Then, $a$ is nonobservable iff for all propositions $p$ in the property and all states $s \in \text{En}(a)$, we have $p(s) = p(a(s))$. The statement $a$ is said to be safe if it is nonobservable and independent from any other statement $b$ for which $\text{Pid}(b) \neq \text{Pid}(a)$.

In the rest of this section we describe in a rather informal way the partial order algorithm from [14]. For the full details about the algorithm we recommend the original references [14, 16].

The reduction of the search space is effected during the DFS, by limiting the search from a state $s$ to a subset of the transitions that are enabled in $s$, the so-called ample set. Such an ample set is formed in the following way: If there is a process $P_i$ who can potentially execute² only safe statements in $s$, and for all transitions $s \xrightarrow{a} s'$ that are labeled with statements from $P_i$ (i.e. $\text{Pid}(a) = i$) the state $s'$ is not on the DFS stack, then the ample set consists of all the transitions from this process only, i.e. all transitions $t$ from $s$ such that $\text{Pid}(\text{Label}(t)) = i$. Otherwise, the ample set consists of all enabled transitions in $s$.

If we now introduce the notion of ample process set, consisting of the processes that have a transition in the ample set, then the reduced DFS algorithm is obtained by replacing in the standard DFS exploration algorithm (Fig. 1) line 4 by the line

² See for more details [14].
for each process $i$ in ample process set do

Obviously the ample process set consists of either only one or all the processes in the system. In the latter case there is no reduction and the reduced search algorithm behaves as the standard DFS, for this particular invocation.

```plaintext
1  proc dfs(s)
2     add s to Statespace
3     /* for each successor $s'$ of s do */
4     for each process $i = 1$ to $N$ do
5         nxt = all transitions enabled in s with $\text{Pid}(t)=i$
6         for all $t$ in nxt do
7             $s' = \text{successor of } s \text{ via } t$
8             if $s'$ not in Statespace then
9                 dfs($s'$)
10            fi
11         od
12     od
13 end
```

**Fig. 1.** Standard depth first search algorithm.

It can be proven [14, 16] that one can use the reduced state space (XLTS) $R(T)$, obtained as described above, instead of the original XLTS $T$ to check any program property which is stated as an LTL formula. In [14, 16] it is shown that

**Theorem 1** ([14, 16]). There exists a reachable acceptance cycle in $T$ iff there exists a reachable acceptance cycle in $R(T)$.

The acceptance cycle in the reduced state space is detected by the cycle-check algorithm in Spin that we consider in more detail in Section 5.1.

The condition that all transitions from the ample set must end out of the DFS stack, the so-called “cycle proviso”, ensures that a statement that is constantly enabled, cannot be “forgotten” by leaving it outside the ample set in a cycle of transitions.

While the cycle proviso is clearly locally checkable during a DFS, the condition that an enabled statement is safe is not, as the definition of safety requires independence from any concurrent statement. A sufficient condition for safety of a statement $a$ that can be checked locally is that $a$ does not touch any global variables or channels. Indeed, it is this condition that is implemented in Spin.

However, it will turn out that one solution for our incompatibility problems will be to refine this safety criterion.

4 The **unless** Construct

The **unless** construct is a mean for modeling exception handling routines and priority choices. Its syntax is $\text{stmt unless stmt}$. The first (left-hand) statement
is called *normal* or *main*, while the second (right-hand) is the *escape* statement.\(^3\) Semantically, the executability of the normal statement depends on the executability of the escape sequence. The escape sequence has higher priority than the normal statement, which means that the normal statement will be executed only if the escape statement is not executable. Otherwise the escape statement is executed and the normal statement is ignored (skipped). This dependence between the two statements of *unless* causes problems when the partial order reduction is used and the escape statement is a rendez-vous communication.

```promela
chan c = [0] of {bit}

active proctype A()
{
    skip; c?1;
}

active proctype B()
{
    assert(false) unless c!1;
}
```

**Fig. 2.** Motivating example for unless statement.

Let us consider the motivating Promela example given in Figure 2.\(^4\) (In the sequel we assume that the reader is familiar with Promela.) Suppose that both A and B are in their starting points, i.e. A is trying to execute its `skip` statement, while B is attempting to do its `unless` statement. Obviously the higher priority rendez-vous send offer c!1 issued by B cannot find a matching receive, so the verifier should detect the assertion violation `assert(false)`\(^5\). However, in the reduced search this is not detected, because of the incorrect partial order reduction. The problem with the reduction occurs because the `skip` statement is not safe anymore. Namely, the criterion that a statement is safe if it does not affect any global objects is no longer true. Because the executability

\(^3\) In general, both statements can be sequences of Promela statements. Also the *unless* construct can be nested. The results from this chapter can be extended in a straightforward way for this general case.

\(^4\) The example is distilled from a model made in the discrete time extension of Spin DTSpin [2]. The model was written by Victor Bos, who first drew our attention to the possible problems with the *unless* statement.

\(^5\) Strictly speaking in this example we are considering a safety property that is not expressed as an LTL formula. The equivalent formulation of the property in LTL can be done in a straightforward way and the partial order reduction will fail because of the same reason as in the present case. We use this version of the example for the sake of simplicity.
of the rendez-vous statement $c?1$ can be changed only because of the change of the location in process A (program counter), no statements are unconditionally globally independent according to the Definition 1.

![Diagram](image)

**Fig. 3.** Interdependence between the *unless* construct and a statement which is safe in absence of *unless*.

The reason is depicted in Fig. 3. In the starting state described above the rendez-vous send $c!1$ is disabled, but with the execution of *skip* it becomes enabled. This means that *skip* has indirectly disabled *assert(false)* which was enabled in the starting state. In that way *skip* and *assert(false)* are not independent according to the Definition 1, because *skip* disables *assert(false)*.

The problem can be solved both statically in compile time or dynamically during the exploration of the state space. The dynamic solution consists of checking whether in a given state there is a disabled rendez-vous statement (more precisely, rendez-vous send) which is part of an escape statement and in that case the partial order reduction is not performed in the given state. The drawback of this solution is that it can be time consuming.

The static solution is to simply declare each statement which is followed in the process specification (*proctype*) by a rendez-vous communication (more precisely, by a rendez-vous receive) as unsafe. We use the term *followed* taking into account all the cycles and jumps in the Promela specification. For example, the last statement of the body of an iteration is followed by the first statement of the body. Whether a given statement is followed by a rendez-vous can be checked by inspecting Spin’s internal representation of the Promela program (abstract syntax tree). This can be done during the generation of the C source (*pan.c*) of the special purpose verifier for the program. Thus, the solution does not cause any time overhead during the verification. Its drawback with regard to the dynamic solution is that the reduction can be less effective because of an unnecessary strictness. It can happen that the reduction is unnecessarily prevented even when in the state that is considered by the DFS there is no
disabled rendez-vous send in an unless construct or even there is no statement with an unless construct at all.

5 Fairness

A pattern very similar to the one from the previous section that causes the partial order reduction algorithm to fail in presence of rendez-vous communications, occurs when the weak fairness option is used in the verification. The weak fairness algorithm is also a very instructive example how things can become complicated because of the feature interaction.

5.1 The Standard Nested Depth-First Search (NDFS) Algorithm

The weak fairness algorithm we are going to deal with is an extension of the nested depth first search (NDFS) algorithm by Courcoubetis, Vardi, Wolper and Yannakakis [6] for memory efficient verification of LTL properties. The algorithm is a more efficient alternative to the usual computation of the strongly connected components of the underlying graph for the XLTS. It is also compatible with Spin’s bit-state hashing, which is not the case with the strongly connected components algorithm. We start with a brief overview of the NDFS algorithm given in Figure 4.

The core idea of the algorithm is to extend the standard DFS of the state space in Fig. 1 with a procedure that checks for an acceptance cycle. Thus, whenever the standard DFS is about to retract from an acceptance state, it is interrupted and the cycle-check procedure is called. The procedure is again a DFS which is started with an acceptance state as a root (seed). If the root state is matched within the cycle-check procedure, an acceptance cycle is reported and the algorithm is stopped. Otherwise, the standard DFS is resumed from the point it has been interrupted. If no cycle is found, than the property is successfully verified.

We need to work with two copies of the state space in order to ensure that the second DFS does not fail to detect a cycle by cutting the search because it has encountered a state visited already by the first DFS. To distinguish between states belonging to different copies we extend the state with one bit denoting the state space copy.

The following theorem from [6] establishes the correctness of the algorithm.

Theorem 2 ([6]). Given an XLTS $T$, when started in the initial state $\hat{s}$, the algorithm in Fig. 4 reports a cycle iff there is a reachable acceptance cycle in $T$.

Note 1. An important feature of the NDFS algorithm is that any state in the second copy of the state space is visited in at most one of the calls to dfs2. This is due to the characteristic position of the call of the cycle-check procedure dfs2, namely, immediately before the recursion retracts from an acceptance state. As shown in [6], this ensures that the part of the second copy of the state space that has been explored by the previous calls of dfs2 can be reused in the current call of the cycle-check procedure.
Fig. 4. Nested depth first search algorithm.
5.2 Description of the Weak Fairness Algorithm

We consider weak fairness with regard to processes, i.e. we say that a given execution sequence is fair if for each process that becomes continuously enabled starting at some point in the execution sequence, a transition belonging to this process is eventually executed. Formally

**Definition 2.** An infinite execution sequence \( s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots \) is fair iff for each process \( P_l, 1 \leq l \leq N \), the following holds: If there exists \( i \geq 0 \) such that \( P_l \) is enabled in \( s_j \) for all \( j \geq i \), then there are infinitely many \( k \geq i \) such that \( P_l \) is executed in \( s_k \).

A cycle \( c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0 \) is fair iff whenever a process \( P \) is enabled in all states \( s_i, 0 \leq i < n \), then \( P \) is executed in some state \( s_j, 0 \leq j < n \).

When model checking under fairness, we are interested only in fair acceptance runs (sequences). This means that we require the detected cycles to be fair, i.e. each continuously enabled process along the cycle contributes at least one transition to it.

The weak fairness (WF) algorithm we are considering here is by Gerard Holzmann and it is a variant of Choueka’s flag algorithm [13]. The weak fairness algorithm is also implemented in Spin.

The basic idea behind the weak fairness algorithm is to apply the NDFS to an extended state space instead of the original one. The extended state space is designed such that to each fair acceptance cycle from the original state space, there is a corresponding acceptance cycle, which need not be fair, in the extended state space, and vice versa. As a consequence, the model checking can be done on the extended state space instead on the original one.

The extended state space consists of \( N + 2 \) copies of the original state space, where \( N \) is the number of processes. To distinguish between the different copies of a same state \( s \), we extend each state with a counter component whose value denotes the copy of the state space the state \( s \) belongs to. Intuitively, when the algorithm operates in a particular copy, this means that it has reached a certain phase. Our goal is to detect a cycle that:

1. contains at least one state which is an acceptance state in the original state space, and
2. along which each process either executes a transition or becomes disabled.

In order to treat the two cases of condition 2 uniformly, we consider that when process \( i \) is disabled, it executes a special \( \epsilon_i \)-transition, where \( 1 \leq i \leq N \), which does not change the state. The \( \epsilon_i \)-transitions are labeled with the special statement \( \epsilon_i \). This statement differs from all other statements and only changes the state counter component to \( i + 1 \).

We check the fulfillment of the two conditions above in a sequential manner. To this end the algorithm passes cyclically through all the copies, from 0 through \( N + 1 \) and back to 0.\(^6\)

\(^6\) As we will see shortly, the NDFS algorithm applied on the extended state space starts in copy 0, while the cycle checks with `dfs2` begin in copy \( N + 1 \). See also the code of the algorithm in Fig. 7.
Schematically, the extended state space is depicted in Fig. 5. The algorithm resides in copy 0 until an acceptance state is encountered, i.e., condition 1 is achieved. When this happens it immediately passes to copy 1. To this end we introduce another type of special transitions, labeled with the special statement $\epsilon_0$. Like $\epsilon_i$-transitions, an $\epsilon_0$-transition too does not change the state, but only the counter (in this case from 0 to 1). The $\epsilon_0$-transitions (statements), however, do not belong to any process.

Copies 1 to $N$ correspond to each of the processes. Intuitively, when the algorithm operates in copy $i$, $1 \leq i \leq N$, it is waiting for process $i$ to execute a transition (ordinary or $\epsilon$). When in copy $i$ process $i$ executes a transition the algorithm passes to copy $i + 1$. Otherwise, if some other process executes a transition, the algorithm stays in copy $i$.

The cycle check always starts in copy $N + 1$. All the states in this copy are considered as acceptance states in the new extended state space. The most important characteristic of the copy $N+1$ is that, unlike in the other copies, there are no transitions from itself leading back to it. More precisely, each transition that generates a state in copy $N+1$ is immediately followed by an $\epsilon_0$-transition to copy 0. This is to avoid detection of acceptance cycles which might be closed by staying inside copy $N+1$ and which might not satisfy conditions 1 and 2.
From the state space structure one can see that each cycle passing through copy $N+1$ must also pass through all the other copies. Therefore, such a cycle satisfies conditions 1 and 2 and corresponds to a fair acceptance cycle in the original state space.

In the sequel we give more formal treatment of the intuitive picture given above. We begin by defining the extended state space:

**Definition 3.** Given an XLTS $T = (S, s, \tau, L, F)$ and processes $P_1, \ldots, P_N$, $N \geq 2$, we define its (weakly) fair extension $F(T)$ to be the smallest XLTS $(S_f, \hat{s}_f, \tau_f, L \cup \{\epsilon_0, \epsilon_1, \ldots, \epsilon_N\}, F_f)$, where $L \cap \{\epsilon_0, \epsilon_1, \ldots, \epsilon_N\} = \emptyset$, satisfying:

- $(\hat{s}, 0) \in S_f$
- 1. if $(s, 0) \in S_f$, $s \not\in F$, and $s' \in \tau(s, a)$ for some $a \in L$, then $(s', 0) \in \tau_f((s, 0), a)$,
- 2. if $(s, 0) \in S_f$, $s \in F$, then $(s, 1) \in \tau_f((s, 0), \epsilon_0)$,
- 3. if $(s, C) \in S_f$, $1 \leq C \leq N$, $s' \in \tau(s, a)$, and $\text{Pid}(a) \neq C$, then $(s', C) \in \tau_f((s, C), a)$,
- 4. if $(s, C) \in S_f$, $1 \leq C \leq N$, $s' \in \tau(s, a)$, and $\text{Pid}(a) = C$, then $(s', C + 1) \in \tau_f((s, C), \epsilon_C)$,
- 5. if $(s, C) \in S_f$, $1 \leq C \leq N$, and $P_C$ is not enabled in $s$, then $(s, C + 1) \in \tau_f((s, C), \epsilon_C)$,
- 6. if $(s, N + 1) \in S_f$, then $(s, 0) \in \tau_f((s, N + 1), \epsilon_0)$.
- $\hat{s}_f = (\hat{s}, 0)$
- $F_f = \{(s, N + 1) \mid s \in S\} \cap S_f$.
- $\text{Pid}(\epsilon_0) = 0$ and $\text{Pid}(\epsilon_i) = i$, for all $1 \leq i \leq N$.

The algorithm which generates and explores the fair extension $F(T)$ is given in Fig. 6. The pseudo-code is an extension of the standard DFS (Fig. 1) that implements Def. 3. Lines 2 and 3 implement case 2 of the definition ($\epsilon_0$-transitions from copy 0 to copy 1). The recursive call of $\text{dfs}$ changes only the counter component. Similarly, lines 5 and 6 implement case 6 ($\epsilon_0$-transitions from copy $N+1$ to copy 0). cases 3, 4 and 5 are implemented in the main part of the algorithm (lines 7–19). The counter increment required in cases 4 and 5 is captured by line 9. If the value of the counter $C$ does not match the process ID $i$ then there is no increment, which is in accord with case 3. Further, case 5 ($\epsilon_i$-transitions) is implemented with lines 11 and 12. The check for $C = i$ in line 12 is to ensure that an $\epsilon_i$-transition is created only if the ID of the disabled process matches the counter component. Finally, case 0 is also implicitly captured. Namely, if the counter component $C$ is 0, but $s$ is not an acceptance state in the original state space, then lines 2 and 3 do not apply, and also there is no increment of the counter component in line 9, because $C$ is 0 and $i$ ranges from 1 to $N$.

An obvious way to check the system under weak fairness is to first generate $F(T)$ from $T$, using the algorithm from Fig. 6 and then apply the NDFS algorithm to $F(T)$. Because of the efficiency reasons we will do these two steps simultaneously, i.e., combine them in the on-the-fly algorithm given in Fig. 7.

The new version of the algorithm is obtained in an obvious way by inserting the lines for cycle check, i.e., applying the standard NDFS scheme from Fig. 4.
Fig. 6. An algorithm for generating the weakly fair extension $F(T)$.

There exists a reachable fair acceptance cycle in $T$ iff there exists a reachable acceptance cycle in $F(T)$.

Proof. Let us denote the sets of transitions of $T$ and $F(T)$ with $R$ and $R_f$, respectively. Let $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n$ be a reachable fair acceptance cycle in $T$. We first prove by induction that for each state $s$ in $T$, if $s$ is reachable, then $(s, C)$ in $F(T)$, for some $C \geq 0$, is also reachable. The induction is on the length of the shortest path(s) between the initial state $\hat{s}$ and $s$. For length 0 the claim trivially holds, because $\hat{s}$ is the only state which is reachable via a path with length 0 and by Def. 3 the state $(\hat{s}, 0)$ is in $F(T)$. Assume that $s$ is reachable from $\hat{s}$ via path of length $n + 1$, $n \geq 0$. Then there exists a state $s'$ reachable through a path of length $n$, such that $s' \xrightarrow{a} s \in R$, for some $a \in L$. By the induction hypothesis there exists $(s', C')$ in $F(T)$ which is reachable. We will prove that there exists an execution sequence leading from $(s', C')$ to $(s, C)$, $C \geq 0$. We have to consider several cases, taking into account Def. 3:

(i) If $1 \leq C' \leq N$, then the execution sequence consists of only one transition which is implied by cases 3 and 4 (and because of the transition $s' \xrightarrow{a} s \in R$).

(ii) If $C' = 0$ and $s'$ is not an acceptance state in $T$, then the desired execution sequence (transition) exists because of case 1.
proc dfs1(s,C) 
  add {s,C,1} to Statespace 
  if C == 0 and acceptance(s) then 
    dfs1(s,1) /* epsilon0 move to copy 1 */ 
  else if C == N+1 then 
    dfs1(s,0) /* epsilon0 move to copy 0 */ 
  else 
    for each process i = 1 to N do 
      if C == i then C' = C+1 else C' = C fi 
      nxt = all transitions t enabled in s with Pid(t)=i 
      if nxt = empty then /* epsilon move */ 
        if {s,C',1} not in Statespace and C == i then dfs1(s,C') fi 
      else 
        for all t in nxt do 
          s' = successor of s via t 
          if {s',C',1} not in Statespace then dfs1(s',C') fi 
        od 
      fi 
    od 
  fi 
  if acceptance((s,C)) then seed:={s,C,1}; dfs2(s,C) fi 
end 

proc dfs2(s,C) /* the nested search */ 
  add {s,C,1} to Statespace 
  if C == 0 and acceptance(s) then 
    dfs2(s,1) /* epsilon0 move to copy 1 */ 
  else if C == N+1 then 
    dfs2(s,0) /* epsilon0 move to copy 0 */ 
  else 
    for each process i = 1 to N do 
      if C == i then C' = C+1 else C' = C fi 
      nxt = all transitions enabled in s with Pid(t)=i 
      if nxt = empty then /* epsilon move */ 
        if {s,C',1} == seed then report cycle 
        else if {s,C',1} not in Statespace and C == i then dfs2(s,C') fi 
      else 
        for all t in nxt do 
          s' = successor of s via t 
          if {s',C',1} = seed then report cycle 
          else if {s',C',1} not in Statespace then dfs2(s',C') fi 
        od 
      fi 
    od 
  fi 
end

Fig. 7. Weak fairness (WF) algorithm.
(iii) If \( C' = 0 \) and \( s' \) is an acceptance state in \( T \), then, by case 2, there exists the transition \((s', 0) \xrightarrow{\epsilon_0} (s', 1)\) and this case reduces to case (ii).

(iv) If \( C' = N + 1 \), by case 6 there exists the transition \((s', N + 1) \xrightarrow{\tau_0} (s', 0)\) and the proof boils down to case (ii) or case (iii).

We construct an acceptance cycle in \( F(T) \) that corresponds to \( c \), by unfolding \( c \) according to the definition of \( \tau_f \) (Def. 3). Without loss of generality suppose that we start at \( s_0 \) by mapping it to \((s_0, C) \in F(T)\) for some \( C \geq 0 \), whose existence we just proved. We continue traversing \( c \) and mapping its transitions to transitions in \( F(T) \) according to the definition of \( \tau_f \). The only ambiguity that might occur is resolved such that case 5 from the definition of \( \tau_f \) has priority over case 3. More precisely, suppose we have to extend the obtained path in \( F(T) \) from some state \((s, C') \in S_f \) by mapping a transition \( s \xrightarrow{a} s' \in R \) and the process \( C' \) is disabled (so, \( Pid(a) \neq C' \)). In this case we first extend the path in \( F(T) \) with \((s, C') \xrightarrow{\tau} (s, C' + 1) \in R_f \) and after that we continue the mapping of \( s \xrightarrow{a} s' \). As \( c \) is fair, after several steps (bounded by \( N(n + 1) \)) we will arrive at some acceptance state \((s_{a_1}, N + 1) \in F(T)\). According to case 6 of Def. 3, from this state we pass via an \( \epsilon_0 \)-transition to \((s_{a_1}, 0)\). After several steps we will encounter an acceptance state from which we pass via a \( \epsilon_0 \)-transition to a state with counter value 1. The counter is increased until again some acceptance state \((s_{a_2}, N + 1) \) is hit. If \( s_{a_1} = s_{a_2} \), then we are done because we have closed the desired acceptance cycle. Otherwise, we repeat the whole procedure from \((s_{a_2}, N + 1)\). As the number of states in \( c \) is finite, we will inevitably revisit some acceptance state \((s_{a_k}, N + 1) \) and in that way generate the corresponding cycle in \( F(T) \).

For the other direction, observe that each acceptance cycle in \( F(T) \) must contain some state \((s, 0)\), such that \( s \in F \) is an acceptance state in \( T \). This is because in \( F(T) \) from an acceptance state we always pass to a state with counter component 0 (Def. 3, case 6). From Def. 3 it is clear that the only transition from copy 0 is via a \( \epsilon_0 \)-transition (case 2), from some state \((s', 0)\) such that \( s' \in F \). Further, it is obvious that in order to get back to the acceptance state in copy \( N + 1 \), one has to pass through all copies from 1 to \( N \), i.e., each process contributes a transition to the cycle, or is disabled in some state. So, each acceptance cycle in \( F(T) \) can be mapped into a fair acceptance cycle in \( T \) by simply removing the \( \epsilon_0 \)- and \( \epsilon_1 \)-transitions and merging accordingly the states that are connected via these. In the same way we can map any path from the initial state \((s, 0)\) to some state \((s, C) \) on the acceptance cycle in \( F(T) \), into a path from \( s \) to \( s \) in \( T \), which gives the reachability of the obtained cycle in \( T \).

As the WF algorithm is in fact the NDFS algorithm applied to \( F(T) \), the correctness of the algorithm is implied by Lemma 1 and Theorem 2, which imply the following theorem.

**Theorem 3.** Given an XLTS \( T \), when started in the initial state \((s, 0)\) of \( F(T) \), the algorithm in Fig. 7 reports a cycle iff there is a reachable fair acceptance cycle in \( T \).
Note 2. The WF algorithm can be optimized by eliminating $\epsilon_0$ and $\epsilon_i$-transitions. In Spin there are no $\epsilon_0$-transitions. Also part of the $\epsilon_i$-transitions are eliminated and all intermediate states that are produced by the $\epsilon_i$-transitions. However, these optimizations can be regarded as an implementational detail which is not relevant for the discussions in this chapter.

5.3 (In)compatibility with the Partial Order Reduction Algorithm

The Motivating Counter Example. One can regard $F(T)$ as an ordinary LTS by abstracting from the state structure. Thus, it is straightforward to show that $F(T)$ is compatible with the partial order reduction techniques in general ([16, 14, 11, 20, 18]), provided that the original set of independent statements in $T$ is adjusted in an appropriate way. More precisely, one has to reduce the set of independent statements because of the $\epsilon_i$-transitions ($1 \leq i \leq N$). Notice that $\epsilon_0$-transitions are not a problem because they are trivially independent of any other action in the system. This is because a $\epsilon_0$-statement is always the only action (transition) from a given state, i.e., it is never simultaneously enabled with any other action (from the Def. 3, cases 2 and 6). The $\epsilon_i$-statements can be dependent with some of the statements that are independent in $T$. In what follows we explain the dependence and show how the compatibility of the WF algorithm with POR in Spin can be repaired by restricting the set of independent statements.

The incompatibility of the weak fairness algorithm with POR (because of rendez-vous communications) was first observed by Dennis Dams on the example in Fig. 8:

In the model from Fig. 8 the LTL formula $\Diamond p$, where $p$ is defined as $b == true$, is not valid even under fairness. The crucial role in the incompatibility is played by the statement $x = 0$ in process $C$. Namely, because of this statement, the rendez-vous send $c!1$ from process $B$ is no longer continuously enabled. Since, now there exists a fair cycle formed just by the statements from $A$ and $C$, i.e., $c!0; c?x; x = 0; c!0 \ldots$ along which the process $B$ can be safely ignored because it is not continuously enabled.

However, when partial order reduction was used the verifier incorrectly reported that the formula was valid. So, the aforementioned fair cycle formed by the processes $A$ and $C$ was not discovered. The reason for the failure is very similar to the one for the unless statements. Again the same pattern occurs of a wrong partial order reduction because of the incorrect independence relation. But this time the problem is related to the $\epsilon_i$-transitions. Recall that those transitions only change the counter component $C$ when all statements of the process $P$, such that $\text{Pid}(P) = C$, are disabled. As illustrated in Fig. 9 the problem is again caused by the rendez-vous statements.

With $s$ we denote the state in the original state space in which process $C$ is about to execute the statement $x = 0$, the processes $A$ and $B$ are hanging on

\footnote{Note that if one tries to run the example with the recent releases of Spin an error will be issued because of the incompatibility of fairness and partial-order reduction in models with rendez-vous operations.}
chan c = [0] of {bit};
bool b = false;

active proctype A()
{
    starta:
        c!0;
        goto starta
}

active proctype B()
{
    startb:
        c!1; b = true;
        goto startb
}

active proctype C()
{
    bit x;
    startc:
        c?x; x = 0;
        goto startc
}

Fig. 8. Motivating example for the fairness algorithm.

their rendez-vous sending statements, and the counter C from the WF algorithm equals the Pid of process B. Then, the statement $x = 0$ is no longer safe, because it is dependent with the statement $\epsilon_{Pid(B)}$ (i.e. $\epsilon_C$). Namely, because $c!1$ is disabled, according to the WF algorithm the $\epsilon_{Pid(B)}$-statement is enabled. After the execution of $x = 0$ the system passes to the state $s'$ in which $c!1$ becomes enabled, and consequently the $\epsilon_{Pid(B)}$-statement is not possible anymore. On the other hand in $s'$, after $x = 0$, $c!1$ becomes enabled and must be included in the fair cycles. Since Spin assumes that $x = 0$ is safe, in the reduced search the verifier never considers the fair cycle in which process B does not contribute a transition, which is, of course, wrong.

Solution to the Compatibility Problem. There is an apparent analogy with the pattern from the unless case. By enabling a rendez-vous statement (transition) ($c!1$ in combination with $c?x$) we are preventing another transition (in this case the $\epsilon_{Pid(B)}$ transition) which is in discord with the independence definition.

As in the case of the unless construct two kinds of solutions are possible. The first solution is static and it is actually the same with the one for the problem
Fig. 9. Interdependence between the “safe” statements and the $\epsilon_i$-statements

with \texttt{unless}. This is not surprising because we have the same problematic reduction pattern which we can avoid exactly in the same way – by declaring as unsafe all statements that are safe according to the standard criteria in Spin, if they are followed by a rendez-vous receive statement.

A dynamic solution which is analogous to the one for the \texttt{unless} case, also looks plausible. In each state we need to check if there is a possibility of an $\epsilon_i$-transition move caused by a rendez-vous communication. The partial order reduction is not performed if this is the case. Unlike in the \texttt{unless} case, this time the time overhead can be much smaller because we have to check the transitions from only one process - the one whose $\textit{Pid}$ equals the counter component $C$.

6 Conclusion

Promela’s \texttt{unless} construct and the weak fairness algorithm are both incompatible with the partial order reduction algorithm when rendez-vous communications are present in the programs. We gave solutions to both problems by proposing a corrected identification of safe statements or changes in the partial order algorithm. It is hoped that the lessons learned from these problems will be helpful to avoid the interference of the partial order with the prospective new features of Spin.\footnote{The author owes this observation to Dennis Dams.}

\footnote{After the publication of the original version of this chapter, we discovered that the same erroneous pattern can cause incorrect results when POR is used in models with priority choice (\texttt{unless} statements) and communications via so called \textit{exclusive send} and \textit{exclusive receive} buffered channels. In some states the communication statements on such channels are also treated as (conditionally) safe (see [14]). One can construct a counter example which is very similar to the one in Fig. 2 which shows that more strict conditions for the safety of send and receive statements on exclusive channels are needed in presence of \texttt{unless} constructs. The solution to this problem proposed...}
A natural task for the future work would be the implementation of the solutions in Spin. As a first step towards the static solution we tested successfully a prototype implementation done by modifying the Spin mechanism for labeling the safe statements [3]. The first main problem was to find the successor of a given statement. The major difficulty in this context was the handling of the various Promela jump constructs (break, goto, etc.). The second more serious problem was the detection whether some channel is a synchronous (rendez-vous) one. We circumvented this by treating all the channels as rendez-vous, i.e., each statement which is followed by a receiving statement on any channel was treated as unsafe. Also the implementation of the first dynamic solution for the fairness should not be too involved.

As a final remark, the compatibility with the weak fairness algorithm can be important for the existing [17, 2] and future extensions of Spin with real time, especially having in mind the work of [4] about Zeno cycles in the real-time systems.

Acknowledgments The author is grateful to Dennis Dams, Gerard Holzmann, Victor Bos and Leszek Holenderski for their valuable information and comments.

References

7. D. Dams, private communication

by the author was implemented by Gerard Holzmann in the standard distribution of Spin (versions 3.3. and later).
4. Integrating Real Time into Spin: A Prototype Implementation

This chapter is a combination of the updated versions of:


and

Integrating Real Time into Spin:
A Prototype Implementation

Dragan Bošnački ⋆ and Dennis Dams

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
PO Box 513, 5600 MB Eindhoven,
The Netherlands,
{dragan, wsindd}@win.tue.nl

Abstract. We present a discrete-time extension of Promela, a high level
modelling language for the specification of concurrent systems, and the
associated Spin model checker. Our implementation is fully compatible
with Spin’s partial order reduction algorithm, which is indeed one of its
main strengths. The real time package is for most part orthogonal to
the other features of the tool, resulting in a modular extension. We have
evaluated it by several experiments, with encouraging results.

1 Introduction

Promela is a high level modelling language for specification of concurrent sys-
tems. The models written in Promela serve as input to the Spin software package
for their automated verification.

The time ordering of actions in a Promela program is implicit and depends
on the (fixed) sequential composition of statements within each one of the com-
ponent processes, as well as on the (unspecified) interleaving of statements from
different processes. This time relation is only qualitative, meaning that we do not
know the exact time interval that will elapse between two events. This can be a
shortcoming when systems are to be verified whose correct functioning depends
on timing parameters. Many such examples can be found among communication
protocols that have to deal with unreliable transport media, where the duration
of timeout intervals is important.

In this chapter, we introduce an extension to Promela that allows to quantify
the time elapse between events, by specifying the time slice in which they occur.
We describe a prototype implementation that integrates this extension into the
Spin tool. Of particular concern is the compatibility of such an extension with the
partial order reduction algorithm, which is an approach to alleviate the state-space explosion inherent in model checking, and indeed one of Spin’s main
strengths. We prove that Spin’s partial order algorithm remains correct under the

⋆ On leave from the Institute of Informatics, Faculty of Natural Sciences and Mathe-
matics, University “Sts. Cyril and Methodius”, Skopje, Macedonia
new timing constructs, and conduct a number of experiments that demonstrate its effectiveness.

The prototype implementation described in this chapter grew out of an earlier implementation of discrete time in Promela and Spin [4] instigated by process algebra ACP [3] and clocked transition systems [18]. There, the timing constructs from the process algebra ACPdrt were modelled by macro definitions entirely on the level of Promela. An advantage of such an approach is that the real-time package thus obtained is orthogonal to Spin: when needed, it can be incorporated as a simple header-file inclusion, for the current and future versions of Spin. Furthermore, the resulting real-time extension is easily moderated, allowing to gain experience with several alternative syntactical constructs and semantic models of time, and also to study the interaction with Spin’s other features, most notably its partial order reduction algorithm. An obvious drawback of such a “high-level” solution is its inefficiency, which is to be avoided particularly in the case of a model checker. The real-time extension of Promela described in the current chapter still uses macro definitions for certain constructs. However, the operation that occurs most frequently, namely the advance of time (“tick”), has been implemented on a lower level, in the source code of the Spin tool.

Eventually, we envisage a more complete integration of the real-time package into Spin, not only regarding the level of implementation, but the integration of real time and partial order reduction as well. Also, the underlying mathematical model will be changed from discrete to dense time. While this complicates the implementation, it is commonly accepted (c.f. [1]) that the resulting model-checking algorithms for dense time have comparable computational complexity. Having said this, we do stress that also discrete-time models are sufficient for a broad range of practical applications [14].

A closely related development is the RT-Spin package of [25]. Although that extension of Spin is more general in that it can model Timed Automata ([1]), with real-valued clocks, it has a number of drawbacks. The most important one is that it is not compatible with the partial order reduction algorithm. Furthermore, Timed Automata lack a notion of urgency; instead, the fact that a transition with a deadline must be taken on time, has to be modelled explicitly in RT-Spin. Also, the package has not been kept up-to-date with Spin versions later than Version 2.0.

The extension of partial order reduction techniques to apply to real-time systems has recently been studied in [21] and [20]. Other model checkers that cover timed behaviour are, e.g., UPPAAL [17], COSPAN [2], PMC [23], HyTech [13]. The latter indeed is able to handle the more general class of linear hybrid systems.

2 The Spin Model Checker

Promela and Spin have been developed for the analysis and verification of communication protocols. The language syntax is derived from C, but also uses the denotations for communications from Hoare’s CSP and control flow statements.
based on Dijkstra’s guarded commands. The full presentation of Promela and Spin is beyond the scope of this chapter. We suggest [15] as a reference to the interested reader; here, we only give a brief overview of Spin’s verification capabilities.

In Promela, system components are modeled as processes that can communicate via channels either by buffered message exchanges or rendez-vous operations, and also through shared memory represented as global variables. The execution of statements is asynchronous and interleaved, which means that in every step only one enabled action is performed, without any assumptions of the relative speed of process executions.

Given as input a Promela model, Spin generates a C program that performs a verification of the system by scanning the state space using a depth-first search (DFS) algorithm. This way, both safety properties such as absence of deadlock, unspecified message receptions, invalid end states and assertions can be checked, as well as liveness properties such as non-progress cycles and eventual reception of messages. The so-called never claims, which are best seen as monitoring processes that run in lock step with the rest of the system, are the most general way to express properties in Spin. Being Büchi Automata, they can express arbitrary omega-regular properties. Spin provides an automatic translator from formulae in linear-time temporal logic (LTL) to never claims. When errors are reported, the trace of actions leading to an invalid state or cycle is saved, so that the erroneous sequence can be replayed as a guided simulation.

For large state spaces methods, Spin features state-vector compressing, partial-order reduction and bit-state hashing.

2.1 Partial Order Reduction

Partial order reduction (POR) is a technique to cope with the state explosion problem in model-checking. We give a brief introduction to the POR algorithm that is used in Spin, rephrasing some definitions from [16].

The basic idea of the reduction is to restrict the part of the state space that is explored by the DFS, in such a way that the properties of interest are preserved. To this purpose, the independence of the checked property from the possible interleavings of statements is exploited. More specifically, two statements \( a, b \) are allowed to be permuted precisely when, if for all sequences \( v, w \) of statements:

\[
\text{if } vabw \text{ (where juxtaposition denotes concatenation) is an accepted behaviour, then } vbaw \text{ is an accepted behaviour as well.}
\]

In practice, sufficient conditions for such permutability are used that can be checked locally, i.e., in a state. For this, a notion of “concurrency” of statements is used that captures the idea that transitions are contributed by different, concurrently executing processes of the system.

The semantics of a Promela program can be represented as a labeled transition system (LTS). An LTS is a triple \( (S, s_0, T) \), where \( S \) is a finite set of states, \( s_0 \) is a distinguished initial state, and \( T \subseteq S \times S \) is a set of transitions. Every transition in \( T \) is the result of executing a statement in some process of the Promela program. We introduce a function \( Label \) that maps each transition to
the corresponding statement. For a statement \( a \), \( \text{Pid}(a) \) denotes the process to which \( a \) belongs and \( \text{En}(a) \) denotes the set of (global) states in which \( a \) is enabled. For \( q \in \text{En}(a) \), \( a(q) \) is the state which is reached by executing \( a \) in state \( q \).

Concurrent statements (i.e. statements with different \( \text{Pids} \)) may still influence each other's enabledness, whence it may not be correct to only consider one particular order of execution from some state. The following notion of independence defines the absence of such mutual influence. Intuitively, two statements are independent if in every state where they are both enabled, they cannot disable each other, and are commutative, i.e., the order of their execution makes no difference to the resulting state.

**Definition 1.** The statements \( a \) and \( b \) are independent iff for all states \( q \) such that \( q \in \text{En}(a) \) and \( q \in \text{En}(b) \),

\[ a(q) \in \text{En}(b) \text{ and } b(q) \in \text{En}(a) \text{, and} \]
\[ a(b(q)) = b(a(q)) \text{.} \]

Statements that are not independent are called dependent.

Note that \( a \) and \( b \) are trivially independent if \( \text{En}(a) \cap \text{En}(b) = \emptyset \). An example of independent statements are assignments to or readings from local variables, executed by two distinct processes.

Another reason why it may not be correct to only consider only one particular order of execution from state \( s \) of two concurrent statements \( a \) and \( b \) is that the difference between the intermediate states \( a(s) \) and \( b(s) \) may be observable in the sense that it influences the property to be checked. For a given proposition \( p \) that occurs in the property (an LTL formula), and a state \( s \), let \( p(s) \) denote the boolean value of the proposition \( p \) in the state \( s \). Then, \( a \) is invisible iff for all propositions \( p \) in the property and all states \( s \in \text{En}(a) \), we have \( p(s) = p(a(s)) \). \( a \) is said to be safe if it is invisible and independent from any other statement \( b \) for which \( \text{Pid}(b) \neq \text{Pid}(a) \).

The reduction of the search space is now effected during the DFS, by limiting the search from a state \( s \) to a subset of the statements that are enabled in \( s \), the so-called ample set. Such an ample set is formed in the following way: If there is a process which has only safe statements enabled and all those transitions lead to a state which is not on the DFS stack, then the ample set consists of all the statements from this process only. Otherwise, the ample set consists of all enabled statements in \( s \). It can be proven \([16, 22]\) that the reduced graph obtained in this way preserves the properties of the original LTS, stated as an LTL formula. The condition that all transitions from the ample set must end out of the DFS stack, the so-called “cycle proviso”, ensures that a statement that it is constantly enabled, cannot be “forgotten” by leaving it outside the ample set in a cycle of transitions.

While the cycle proviso is clearly locally checkable during a DFS, the condition that an enabled statement is safe is not, as the definition of safety requires independence from any concurrent statement. A sufficient condition for safety
of a statement \( a \) that can be checked locally is that \( a \) does not touch any global variables or channels. Indeed, it is this condition that is implemented in Spin.

3 Introducing Discrete Time in Promela and Spin

3.1 Real Time in Promela

In the discrete-time model, time is divided into slices of equal length, indexed by natural numbers. The actions are then framed into those slices, obtaining in that way a measure for the elapsed time between events belonging to different slices. The elapsed time between events is measured in ticks of a global digital clock that is increased by one with every such tick. Within a slice however, we only know the relative ordering between events, as in the time free case. The passage of time has the lowest priority – time can only progress if all the other processes of the system have finished the execution of their actions scheduled for the current time slice.

The example in Fig. 1 illustrates the time model. The elapsed time between

\[ A \text{ and } B \text{ which happen in the } i\text{-th time slice and the events } C \text{ and } D \text{ which belong to } i+3\text{rd time slice is three ticks. However we cannot measure the time distance between } A \text{ and } B, \text{ or between } C \text{ and } D \text{ – we only know that } A \text{ precedes } B \text{ and that } C \text{ is before } D. \]

In state space enumeration methods like the one used in Spin, verification can be regarded as checking all possible simulations of the system. In that sense the simulation can be considered as a primitive for the validation, so we start with the description of the discrete time implementation for simulation purposes. The basic idea is to execute the system time slice by time slice. Recalling that the execution of statements in Spin is asynchronous and interleaved, the basic problem is how to avoid interleaving of actions belonging to different time slices. One way to solve this is by forcing each process to stop the execution after it has executed all the actions for the current time slice, waiting to receive a signal that the system has passed to a new time slice. This synchronization is done by a special “daemon” process that is not a part of the modelled system and waits in the background to become active only when all the other processes from the system are blocked. This daemon process is a pacemaker that transfers the system to the next time slice, by sending unblocking signals to the other processes.

We implement this synchronization scheme by extending Promela with a new variable type \texttt{timer} corresponding to discrete time countdown timers, three new
Integrating Real Time into Spin: A Prototype Implementation

statements \texttt{set}, \texttt{expire} and \texttt{tick} that operate on the new type, and a special timing process \texttt{Timers} which is the daemon process that uses \texttt{ticks} to decrease the timer values. The implementation can be done entirely on user level, without any additional changes in the Spin source code, for example, with the following Promela macro definitions and timer process:

\begin{verbatim}
#define timer int
#define set(tmr,val) (tmr=val)
#define expire(tmr) (tmr==0) /*timeout*/
#define tick(tmr) if :: tmr>=0 -> tmr=tmr-1 :: else fi

proctype Timers()
{ do :: timeout -> atomic{ tick(tmr1); tick(tmr2) } od }
\end{verbatim}

The first macro defines \texttt{timer} as a synonym for the integer type. The \texttt{set} macro sets the value of the timer \texttt{tmr} to \texttt{val}. The \texttt{expire} macro is a test which becomes true when \texttt{tmr} becomes 0. The \texttt{tick} macro is used only in the \texttt{Timers} process and it decreases the value of \texttt{tmr} provided that it is active, i.e., its value is non-negative. Timers with negative values are considered as deactivated. \texttt{Timers} consists of an endless \texttt{do} iteration that realizes the passage of time. It is run concurrently with the other processes of the system. The key idea of the concept is the usage of \texttt{timeout} - a predefined Promela statement that becomes true when no other statement within the system is executable. By guarding \texttt{ticks} with \texttt{timeout}, we ensure that no process will proceed with an action from the next time slice until the other processes have executed all actions from the current time slice.

Within a process, statements are divided into time slices by putting \texttt{set} and \texttt{expire} at the beginning and end, respectively, of each time slice. For instance, assuming that we have declared \texttt{timer tmr}, and that \texttt{A}, \texttt{B} and \texttt{C} are nonblocking Promela statements, the sequence

\begin{verbatim}
set(tmr,1); A; B; expire(tmr); C
\end{verbatim}

means that \texttt{A} and \texttt{B} will be executed in same time slice, while \texttt{C} belongs to the next time slice. The \texttt{expire} statement is a synchronization point where the process waits for \texttt{tmr} to become zero. This can be done only by \texttt{Timers}, i.e., only when all active timers are decreased. Thus, it is guaranteed that \texttt{C} will be executed in the next time slice. In fact, \texttt{set} is only a labelling of the time slice (some sort of reading of the global digital clock we assumed in our time model) and it can be permuted with the statements from the time slice that it labels (in our case \texttt{A} and \texttt{B}).

Also, in cases where we have empty time slices, we optimize sequences of the form

\begin{verbatim}
set(tmr,1); expire(tmr); set(tmr,1); expire(tmr);...; set(tmr,1); expire(tmr);
\end{verbatim}

1 There is one \texttt{tick} for each timer in the system. In the example above we assumed that there are only two timers, \texttt{tmr1} and \texttt{tmr2}. The \texttt{ticks} are wrapped in \texttt{atomic} statement in order to avoid unwanted unblocking of some of the system processes before all timers are decreased.
by \texttt{set(tmr,val); expire(tmr)}, where \texttt{val} is the number of \texttt{set-expire} pairs.

It is often convenient to use derived macros for modeling various delays. For instance, one tick delay and unbounded nondeterministic delay are implemented, by the following macros

\begin{verbatim}
#define delay(tmr,val) set(tmr,val); expire(tmr)
#define udelay(tmr) do :: delay(tmr,1) :: break od
\end{verbatim}

In the unbounded nondeterministic delay, at each iteration a nondeterministic choice is made whether the loop will be broken and the process will proceed with the execution of a new statement, or the decision will be delayed for the next time slice. In a similar way a nondeterministic bounded delay up to a certain number of ticks, or a nondeterministic delay within lower and upper bounds can be modeled.

The expressiveness of our extended Promela is the same as the one of timed automata interpreted in integral time (fictitious clocks, using the terminology of [1]). It is straightforward to show that using the aforementioned derived timing constructs one can model timed automata interpreted in integral time. Conversely, following the approach of [25], the semantics of the extended Promela can be given via timed automata.

It is noteworthy that, because of the independence of the timing implementation from the other parts of the Spin’s source code, the expressivity of the timing framework is in fact augmented with the introduction of new features in Spin. For instance, a scheduler for several processes running on a single processor can be modeled in a natural way like an additional master process that synchronizes the other processes by giving them execution permission via Promela’s \texttt{provided} declarator mechanism, which was recently introduced into Spin (from version 3.0).

\section*{3.2 Example}

In this section we show how the discrete time can be used for the specification and verification of the Parallel Acknowledgment with Retransmission (PAR) protocol [24]. The choice of PAR was motivated by the fact that it is a relatively simple protocol, yet it is complex enough that its correct functioning depends on the duration of time intervals in a nontrivial way. PAR has been used in similar studies as this one, cf. [26, 19].

PAR is one-way (simplex) data-link level protocol intended to be used over unreliable transmission channels which may corrupt or lose data. There are four components in our implementation: a sender, a receiver, data channel K and acknowledgment channel L (Fig. 2). The sender receives data from the upper level and sends them labeled with a sequence number that alternates between 0 and 1 over the channel K. After that it waits for an acknowledgment via the channel L. If this does not occur after some period of time, the sender times out and resends the old data. If the data are received undamaged and labelled with
the expected sequence number, the receiver delivers them to the upper level and sends an acknowledgment.

Of crucial importance here is the duration of the time-out period which should be longer than the sum of the delays through the channels and the message processing time by the receiver. A premature timeout can cause the loss of a frame by the following scenario: The sender sends a frame and times out too early. This causes a duplicate of the just sent frame to be sent too. The receiver receives both frames and sends two acknowledgments. When the sender gets the acknowledgment for the first frame it thinks that it is for the second one (the duplicate). After receiving this first acknowledgment it sends next frame, which is lost by the data channel. In the meantime the second acknowledgment arrives (which was sent by the receiver as a result of successful receiving of the duplicate) and the sender mistakenly thinks that it is an acknowledgment for the last frame and never resends it.

The complete listing of the discrete time Promela model of PAR is given below:

```plaintext
/*discrete time macros*/
define timer int
define set(x,y) x=y
define expire(x) (x==0)
define tick(x) if :: x>=0->x=x-1; :: else; fi
define on(x) (x!=-1)
define delay(x,y) set(x,y); expire(x);
define udelay(x) do :: delay(x,1) :: break od

/*PAR time parameters*/
define dK 3 /* delay along channel K */
define dL 3 /* delay along channel L */
```
'\#define dR 1 /* receiver's message processing time */
\#define To 9 /* sender's timeout value */
\#define MAX 8 /*max number of different message contents*/

/*timers*/
timer sc, rc, xk, xl;

proctype Timers()
{
    do
        :: timeout ->
            atomic{tick(sc); tick(xk);
                tick(rc); tick(xl);} 
    od;
}

/*channels*/
chan A = [0] of {byte, bit};
chan B = [0] of {byte, bit};
chan C = [0] of {bit};
chan D = [0] of {bit};

/*protocol entities*/

proctype Sender()
{
    byte mt; /* message data */
    bit sn=0; /* sequence number*/

    R_h:
        /*message can be sent in any time slice*/
        udelay(sc);
        mt = (mt+1)\%MAX;

    S_f:
        A!mt,sn; /*sand through channel K*/
        set(sc,To);

    W_s:
        do
            :: D?_ ->
                if
                    :: atomic{skip; delay(sc, 1); sn=1-sn; goto R_h;};
                    :: atomic{printf("MSC: ACKerr\n"); goto S_f};
                fi;
            :: expire(sc) -> goto S_f; /*timeout*/
        od;
}

proctype Receiver()
{
byte mr, me=1; /* received and expected message*/
bit rsn, esn=0; /*received and expected sequence number*/

W_f:
    B?mr,rsn;
    if
    :: rsn == esn -> goto S_h; /*correct message and seq. num */
    :: rsn == 1-esn -> goto S_a; /*correct message, wrong seq. num */
    :: atomic{printf("MSC: MSGerr\n");
        goto W_f;};
    fi;
S_h:
    /*Out!mr*/
    assert(mr == me);
    atomic{delay(rc,dR);
        /*message processing delay*/
        esn = 1-esn; me = (me+1)%MAX};
S_a:
    C!1; /*send ack through channel L*/
    atomic{delay(rc, 1); goto W_f;};
}

/*channel processes*/

proctype K()
{
    byte rd; /*received chunk*/
    bit rab; /*received and expected bits*/

    do
    :: A?rd,rab;
        delay(xk, dK);
        if
        :: skip; B!rd,rab;
        :: skip;
        fi;
    od;
}

proctype L()
{
    do
    :: C?_;  
        delay(xl,dL);
        if
        :: skip; D!1;
        :: skip;
        fi;
    od;
}
init
{
    atomic{set(sc,-1); set(rc,-1);
    set(xk,-1); set(xl,-1);
    run Timers();
    run Sender();
    run K();
    run L();
    run Receiver();}
}

The Promela model starts with the already described preamble of macros for discrete time.

Then, after the definition of protocol specific parameters, the timers are declared. There is one timer per protocol entity, i.e., for sender (sc), receiver (rc), channel K (xk) and channel L (xl). Although used locally in this model, the timers are declared on global level in order to be accessible for Timers. The four defined Promela channels A, B, C and D have capacity 0 meaning that they are synchronous rendez-vous channels. A, B, C and D are used only as auxiliary channels, because the “real” unreliable channels are modeled as proctype, i.e. as a separate processes. The channels A and B are the input and output, respectively, to the channel K.

They deal with messages consisting of two fields — one of type byte and one of type bit, corresponding to the message contents and the alternating bit. Similarly, C and D are the input and output for L and they can carry messages containing just one bit — the acknowledgment.

After the declarations of local variables, the Sender process begins with an unbounded start delay of the first operation which fetches a message from the upper level (user). In our model this is simplified by allowing Sender to generate the message itself by the statement \( \texttt{mt = (mt+1)} \mod \texttt{MAX} \). The message followed by a sequence number is sent through the channel K. This event is “labeled” by an activation of the timer sc which is needed for the usual timeout mechanism (not to be confused with the timeout Promela statement) for unblocking the protocol in case of message or acknowledgment loss. After sending the message the Sender blocks waiting for the arrival of an acknowledgment or the expiration of the timeout timer. In the first case a nondeterministic choice \(^2\) is made between signalling an error because of a corrupted acknowledgment and accepting the acknowledgment as correct.

Receiver starts by waiting for a message. This is achieved by the statement \( \texttt{B?mr,sn} \), which denotes the reception of a message through the channel B. The receiving statements in our implementation are treated as executable as soon as possible, i.e. in the same slice when the corresponding sender is ready to send. After the label \( \texttt{S,sh} \) an assert is used to ensure that the received message always matches the expected one. (Note that this statement is added to the model only

\(^2\) The if statement is similar by its semantics to the do statement - a random choice is made between the alternatives and if none of them is enabled, then the statement is blocked.
Integrating Real Time into Spin: A Prototype Implementation

The message processing latency is modeled by a delay on the Receiver’s timer. Also the expected message and sequence number are updated. Successful reception is confirmed by sending a message through L (via Promela channel C) and after a latency delay the Receiver starts waiting again for a new message.

Channel processes model the lossy channels K and L. Both processes have the same structure — after a delay they nondeterministically chose between delivering or losing the message. The skip is always executable. It is used here to give equal precedence for both choices.

All processes are started from a special process called init via run statements. By enclosing the run statements within an atomic it is ensured that they start simultaneously.

Verification of the model can be done like in the standard Spin. Among the other properties, the validator is able to find out and display the scenario of losing a message, in case of an erroneous combination of the timing parameters.

3.3 Low Level Implementation

In order to achieve more efficiency we get down from the high-level implementation to a lower level, by realizing the tick construct in the Spin’s source code. Also, the new semantics of tick is adopted so as to decrease all the timers in the system together. The implementation of the new statement in Spin is completely modular and does not interfere with other features.

The timing procedure now becomes much simpler:

```
proctype Timers
{ do :: timeout -> tick od }
```

As it is universal for all models, it can be made part of the standard header file that contains the basic macros. Another advantage of the new tick is that the timers can be declared locally. Like ordinary local variables, local timers can be dynamically created and deleted together with the processes they belong to. To the best of our knowledge this possibility is not present in any tool for validation of timed systems. A variable number of timers automatically means a smaller state space, due to Spin’s variable size state representation.

The most important benefit of the local usage of timers is the partial order reduction which is, as already emphasized, one of the main strengths of enumerative methods. The integration of partial order reduction with the discrete time model is one of the main results of this chapter and it is the topic of the next subsection.

3.4 Compatibility with Partial Order Reduction

In this section we prove that the partial order reduction algorithm of (standard, untimed) Spin is correct for the timed extension as well. This algorithm is based on selecting, during the DFS exploration of the state space, only statements from the ample set. The construction of an ample set is based on the notion
of safety of statements: given a criterion to determine which statements are safe, the underlying theory, that was presented in Section 2.1, guarantees that the reduction strategy is correct, i.e. that (LTL) properties are preserved. For untimed Spin, this criterion is that a statement does not read or write any global variables or channels. We will now show that this criterion is still sufficient in the presence of timed statements.

**Lemma 1.** Any statement that is safe in standard, untimed Promela, remains safe in the presence of the new timed statements `set`, `expire`, and `tick`. Furthermore, when applied to local\(^3\) timers, the statements `set` and `expire` are safe, and `tick` is independent from any other statement.

*Proof.* In order to show that a statement is safe, we have to show that it is invisible, and independent from any concurrent statement (i.e. any statement with a different `Pid`).

Let \(a\) be a statement in untimed Promela that is safe. Then it is clearly still invisible in timed Promela. Also, \(a\) is obviously still independent from any concurrent untimed statement. \(a\)'s independence from any ("new") timed statement that is concurrent follows from the fact that the set of timer variables, that are the only variables which can be read or written by timed statements, is disjoint from the set of ordinary variables which can be affected by \(a\).

Next, we let \(a\) be one of the statements `set` and `expire`, applied to a local timer. \(a\) is invisible because the local timer variable may not occur in the property to be checked. Let \(b\) be a concurrent statement different from `tick`. Then \(a\) is obviously independent from \(b\) because of the disjoint sets of variables they operate on. Independence between `tick` and \(a\), and indeed independence between `tick` and any timed or untimed statement, follows from the fact that `tick` can never be enabled simultaneously with another statement. This is because of the way `tick` is implemented: it is only enabled in the case of a `timeout` condition.

As `tick` is implemented in the Spin source code, it is not subject to reduction by the partial order algorithm. However, this does not have any repercussions: any `tick` statement, regardless whether it is acting on a local or on a global timer, is the only statement that is enabled, when it is enabled at all. So, no reduction would have been possible anyway.

### 3.5 Zero Cycles

In the semantics of time that we use the flow of time can be stopped if the system enters a so called zero cycle. A zero cycle is a cycle in the state space that consists of a sequence of events which does not contain the `tick` statement. As all the other statements in the system have priority over `timeout`, i.e. `tick`,

\(^3\)Recall that a local timer is a timer variable that is declared local to a Promela process. In timed Spin, it can be manipulated by the declaring process or by the `tick` operation.
the effect of such a cycle is that the `Timers` procedure is never activated and time does not pass.

If one is not careful, it is quite easy to introduce zero cycles in the specifications. Our experience with the SDL specification of the MASCARA protocol (which we revisit below) confirms that.

Naturally, we would like to avoid these artificial situations when the passage of time is blocked. To this end we have to be able to check for existence of zero cycles before we start the verification. We can do this by checking the system for the LTL formula \( \Box \Diamond \text{timeout} \). The latter is a formal equivalent of the claim that along each infinite execution sequence of the system the predicate `timeout` holds infinitely often, i.e., infinitely often the `tick` statement is executed, which implies that time progresses.

## 4 Experimental Results

We have tested the implementation on various models known in the literature (e.g. Train Gate Controller, Seitz Circuit, Leader Election Protocol). We focus our attention on three of them that illustrate the effectiveness of the implementation and in particular the partial order reduction: Fischer's mutual exclusion protocol, the already presented PAR protocol and the Bounded Retransmission Protocol (BRP). We also applied the discrete-time extension to the MASCARA protocol – a telecommunication protocol developed by the WAND (Wireless ATM Network Demonstrator) consortium [9]. Besides partial order reduction we used as an additional option minimized automata, a technique for reduction of the state space recently included in the standard Spin distribution. In the `options` column of the tables below, “n”, “r” and “ma” denote verifications without POR, with POR, and with POR together with minimized automata, respectively.

The version of the Fischer’s protocol that was verified is a translation of the same model written in Promela with real (dense) time of [25], with virtually the same timing parameters. The obtained results for the verification of the mutual exclusion property are shown in Table 1 (N is the number of processes).

As expected, the state space growth is exponential and we were able to validate the model without using POR up to 4 processes. For 6 processes even POR was not enough and it had to be strengthened with minimized automata. Nevertheless, the profit from POR is obvious and even becomes more evident as the number of processes grows. While for \( N = 2 \) the number of states is reduced to 72% compared to the case without POR, and the transitions are reduced to 56%, for \( N = 4 \) the reduction increases to 27% and 12% for states and transitions, respectively.

It is difficult to compare our implementation with the one from [25], because they are based on different time models. Nevertheless, having in mind that the property that was checked was a qualitative one, for which discrete time suffices [14], one can safely say that after \( N = 4 \) our implementation has better performance. In fact, for \( N = 5 \) the validator from [25] runs out of memory. Ob-
Previously, POR is the decisive factor, because without POR our implementation is also incapable to handle cases for $N > 4$.

In the case of the PAR protocol the reduction is very small and even diminishes with the increase of parameters (Table 2).

The insensitivity of PAR model to reduction can be explained by observing that PAR is in fact a sequential algorithm. The protocol entities take turns during the execution of the protocol, most of the time only one of them being active while the others are waiting passively for some trigger-event. Very little concurrent activity results in a bad behaviour of the POR algorithm which deals with concurrent actions from different processes. It is an opposite situation compared to Fischer’s protocol where the higher degree of concurrency led to an improvement of the effects of POR.

Note the sensitivity of the state space to the increase of parameter values. This sensitivity to parameters is analogous to the same effect observed in the

---

**Table 1.** Results for Fischer’s protocol.

<table>
<thead>
<tr>
<th>N</th>
<th>option</th>
<th>states</th>
<th>transitions</th>
<th>memory [MB]</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>n</td>
<td>528</td>
<td>876</td>
<td>1.453</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>378</td>
<td>490</td>
<td>1.453</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>ma</td>
<td>378</td>
<td>490</td>
<td>0.342</td>
<td>0.2</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>8425</td>
<td>10536</td>
<td>1.761</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>3813</td>
<td>4951</td>
<td>1.596</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>ma</td>
<td>3813</td>
<td>4951</td>
<td>0.445</td>
<td>0.3</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>128286</td>
<td>373968</td>
<td>7.085</td>
<td>15.5</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>34157</td>
<td>44406</td>
<td>3.132</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>ma</td>
<td>34157</td>
<td>44406</td>
<td>0.650</td>
<td>33.7</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>288313</td>
<td>377032</td>
<td>17.570</td>
<td>36.2</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>288313</td>
<td>377032</td>
<td>1.059</td>
<td>332.5</td>
</tr>
<tr>
<td></td>
<td>ma</td>
<td>2.35e6</td>
<td>3.09e6</td>
<td>2.118</td>
<td>624.6</td>
</tr>
</tbody>
</table>

**Table 2.** Results for the PAR protocol.

<table>
<thead>
<tr>
<th>time parameters</th>
<th>dK</th>
<th>dL</th>
<th>dR</th>
<th>To</th>
<th>option</th>
<th>states</th>
<th>transit. mem. [MB]</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>n</td>
<td>1318</td>
<td>1822</td>
<td>1.453</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>1116</td>
<td>1533</td>
<td>1.493</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>10</td>
<td>90</td>
<td>90</td>
<td>n</td>
<td>7447</td>
<td>9994</td>
<td>1.761</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>7295</td>
<td>9705</td>
<td>1.801</td>
</tr>
<tr>
<td>300</td>
<td>300</td>
<td>100</td>
<td>900</td>
<td>900</td>
<td>n</td>
<td>68737</td>
<td>91714</td>
<td>5.135</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>68585</td>
<td>91425</td>
<td>5.420</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ma</td>
<td>68585</td>
<td>91425</td>
<td>2.221</td>
</tr>
</tbody>
</table>
region automaton implementation of the dense time system based on timed automata \[1, 2\]. At least for the type of protocols like PAR, when the sequential component of the system is predominant, this problem can be solved by grouping together consecutive ticks into one tick to avoid generation of spurious intermediate states.

The BRP protocol is a simplified version of an industry protocol used by Philips in remote control devices. Our implementation of the protocol is based on \[8\], where the protocol was treated using a combination of Uppaal (for timing aspects) and Spin (for consistency of the specification). We give the results for safety properties (absence of deadlock and assertions). Using the same parameters as in \[8\], all the properties verified there can be checked with a memory consumption of at most 5 Mbytes and less than a minute of CPU time. This is a much better result than the one in \[8\]. In addition, our model has the advantage that the validation is done completely within Spin.

<table>
<thead>
<tr>
<th>option</th>
<th>states</th>
<th>transitions</th>
<th>memory [MB]</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>32967</td>
<td>71444</td>
<td>4.116</td>
<td>6.7</td>
</tr>
<tr>
<td>r</td>
<td>12742</td>
<td>14105</td>
<td>2.517</td>
<td>3.2</td>
</tr>
<tr>
<td>ma</td>
<td>12742</td>
<td>14105</td>
<td>1.571</td>
<td>26.4</td>
</tr>
</tbody>
</table>

Table 3. Results for the timed version of BRP.

<table>
<thead>
<tr>
<th>option</th>
<th>states</th>
<th>transitions</th>
<th>memory [MB]</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>199454</td>
<td>658852</td>
<td>7.515</td>
<td>37.6</td>
</tr>
<tr>
<td>r</td>
<td>113566</td>
<td>294546</td>
<td>5.547</td>
<td>19.0</td>
</tr>
</tbody>
</table>

Table 4. Results for the untimed version of BRP.

The benefit of POR reduction is obvious — the number of states is reduced to 39%, while the transitions are reduced to 20%. In order to get an indication how well the partial order reduction combines with our time extension in Spin, we have compared the above reductions for the BRP to the partial order reduction that is achieved on a slightly different version of the protocol that does not involve timers (see e.g. \[11\] or \[10\]). The result is given in Table 4. The effect for the timed version turns out to be even better: 39% and 20% versus 57% and 45%. It has to be said though that in order to model the effect of timeouts in the untimed case, we need an additional global variable, which may influence the effects of the reduction.

The MASCARA protocol is an extension of the ATM (Asynchronous Transfer Mode) networking protocol to wireless networks. Our model of the protocol consists of one static access point (AP) which communicates with several mobile terminals (MT). A more detailed description of the protocol can be found in
Chapter 5 of this thesis. Tables 5 and 6 contain the results of deadlock check for the case of one and two MTs, respectively.

Table 5. Results for MASCARA with one mobile terminal.

<table>
<thead>
<tr>
<th>option</th>
<th>states</th>
<th>transitions</th>
<th>memory [MB]</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>17421</td>
<td>61514</td>
<td>4.712</td>
<td>4.6</td>
</tr>
<tr>
<td>r</td>
<td>8586</td>
<td>18293</td>
<td>3.106</td>
<td>1.5</td>
</tr>
</tbody>
</table>

For the case with one mobile terminal the the reduction in the number of states is to 50% and in the number of transitions to 30% of the corresponding figures produced without POR. Similar remarks hold for the verification time.

Table 6. Results for MASCARA with two mobile terminals.

<table>
<thead>
<tr>
<th>option</th>
<th>states</th>
<th>transitions</th>
<th>memory [MB]</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>&gt; 4.99 × 10^8</td>
<td>&gt; 2.67 × 10^9</td>
<td>&gt; 20550</td>
<td>&gt; 300000</td>
</tr>
<tr>
<td>r</td>
<td>7.24418 × 10^7</td>
<td>2.67684 × 10^8</td>
<td>3525.314</td>
<td>6754.172</td>
</tr>
</tbody>
</table>

Without POR option for the case with two MTs the machine ran out of memory. The reduction both in the number of states and transitions is at least one order of magnitude.

5 Conclusions and Future Work

We presented an extension of Promela and Spin with discrete time, using integer variables as timers to stamp the time slices. The core idea was to use a special background daemon process to implement the passage of time. For this purpose we used the Promela `timeout` predefined statement as a mechanism for process (i.e. timer) synchronization. We first showed how the concepts can be implemented using only the existing features of Promela and Spin, by extending the language with new time statements as Promela macro definitions. The usage of timing features was demonstrated on the specification and validation of the PAR protocol.

As one step further toward a more efficient integration of the timing framework into Spin, we implemented timing on the level of the validator’s source code. With this low level implementation, we gain the possibility to use the timers locally. This allows for dynamic change of the number of timers in the system — a feature which to the best of the authors’ knowledge does not exist in any implementation of verification of real-time systems. This is in accord with the possibility of dynamic creation of processes already present in Spin. But, the most important benefit of the local usage of timers, and one of the main results
of this chapter, was the adaptation of the existing partial order reduction algorithm to the discrete time setting. The urgency that is a necessary feature in the modeling of a broad class of interesting timing dependent systems is achieved in a natural way, without usage of invariants. The low level implementation is a seamless and completely compatible extension of the standard Spin validator which allows one to use all the existing features of the untimed validator in a standard way. We showed the efficiency of the partial order reduction and the implementation in general on several examples, of which we emphasize the verification of the Bounded Retransmission Protocol and the MASCARA protocol, both originating from industry. The main future tasks will certainly be to test the approach and accumulate experience by doing new verifications on systems known in the literature or some new real-world (industry) cases which are time dependent.

Besides the improvement of the existing prototype, the main direction for the future work will be the extension of Promela and Spin with dense time, which will provide the possibility to express and verify a more general class of properties. The full description of the implementation will be presented in a forthcoming paper. Here we give an outline of the basic idea, assuming that the reader is familiar with the theory of timed automata (see for instance [1]). In the dense-time extension we consider each Promela process in the system as a timed automaton, thus, the system as a whole is then represented as a parallel composition of timed automata. For this purpose we have to extend Promela with a new variable type clock whose value can be tested (compared) and/or set to zero. We are going to implement the discretizations of the timed automata, called region automata. Once we have these discretizations we can reuse the same basic concepts from the discrete-time implementation presented in this chapter. The key idea is that to represent the time passage for the dense time one can again use a discrete global clock which ranges over clock regions (instead over integer valued vectors, as it was for the discrete time). The number of regions is always finite [1] which guarantees termination of the validation procedure. Thus, the whole concept of a special procedure that implements the time passage can be applied, like for the discrete-time case. Although there are many technical details that we omit because of the space limitations, it is obvious that the same concept of timeout as a clock tick can be also reused to obtain an extension which can be easily integrated with the other features of Spin. With regard to the efficiency it is promising that the partial order reduction algorithm can be adapted in the same way as for discrete time [6].

It would be very interesting to incorporate the ideas about data and time abstraction of [7] both in the existing discrete time prototype, as well as in the future dense time implementation, in order to obtain verifications that are independent of the concrete parameter values, and to see how general is their applicability.

Another promising idea is to extend the concept of a time managing process to a class of hybrid systems called discrete time rectangular automata [12]. This can be done by introducing, besides timers (clocks), new time dependent
variables and allowing the timing process to change their value also with steps greater than one [5].

Acknowledgments. The authors would like to thank Stavros Tripakis, Bart Knaack, and the anonymous referees for their helpful comments.

References

4. D. Bošnački, Implementing Discrete Time in Promela and Spin, International Conference on Logic in Computer Science, LIRA ’97, University of Novi Sad, Yugoslavia, 1997. (Copy also available from the author)
Integrating Real Time into Spin: A Prototype Implementation


5. Model Checking SDL with Spin

This chapter was previously published as:

Model Checking SDL with Spin

Dragan Bošnački\textsuperscript{1}, Dennis Dams\textsuperscript{2}, Leszek Holenderski\textsuperscript{1}, and Natalia Sidorova\textsuperscript{2}

\textsuperscript{1} Dept. of Computing Sci., Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
\{D.Bosnacki,L.Holenderski\}@tue.nl

\textsuperscript{2} Dept. of Electrical Eng., Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
\{D.Dams,N.Sidorova\}@tue.nl

Abstract. We present an attempt to use the model checker Spin as a verification engine for SDL, with special emphasis put on the verification of timing properties of SDL models. We have extended Spin with a front-end that allows to translate SDL to Promela (the input language of Spin), and a back-end that allows to analyse timing properties. Compared with the previous attempts, our approach allows to verify not only qualitative but also quantitative aspects of SDL timers, and our translation of SDL to Promela handles the SDL timers in a correct way. We applied the toolset to the verification of a substantial part of a complex industrial protocol. This allowed to expose several non-trivial errors in the protocol’s design.

1 Introduction

We present an approach to automating the formal verification of SDL, by model checking SDL specifications with Spin. SDL \cite{8} is a visual specification language, especially well suited for communication protocols, and quite popular in industry. Spin \cite{5} is one of the most successful enumerative model checkers.

In order to connect the Spin verification engine to SDL, we had to extend Spin in two ways. First, we had to implement a front-end which would allow to automatically translate SDL to Promela (the input language of Spin). Second, we had to extend Spin with the notion of discrete time, to be able to model SDL timers. The extended version is called DT Spin and its input language is called DT Promela (where DT stands for discrete time).

The translation of SDL to Promela is split into two steps. In the first step we use the \texttt{sdl2if} tool, implemented in Verimag, Grenoble, which transforms SDL programs to the intermediate format IF \cite{3} that was designed for the representation of timed asynchronous systems. This first step flattens the hierarchic structure of SDL blocks to bare processes which can then be directly transformed to Promela processes, in the second step, by our tool \texttt{if2pml}.

\footnote{This research has been supported by the VIRES project (Verifying Industrial Reactive Systems, Esprit Long Term Research Project \#23498).}
We applied our method to the verification of a substantial part of MAS- 
CARA which is a complex telecommunication protocol developed by the WAND 
(Wireless ATM Network Demonstrator) consortium [13]. As a result, we exposed 
several non-trivial errors in the design of MASCARA.

In order to resolve the usual problems caused by the lack of the formal 
semantics of SDL, we decided to rely on the semantics of SDL as determined 
by the ObjectGEODE tool [11]. In particular, we assume that transitions are 
atomic and instantaneous, and timeout signals are not necessarily sent at the 
beginning of a time slice (in other words, the timer messages are treated like other 
messages, without any special priority). More details are given in Section 3.2.

We are aware of two other attempts to use Spin as a verification engine 
for SDL [6, 10]. In our opinion, they were not fully successful. First, both ap- 
proaches tackle the qualitative aspects of SDL timers only, in the sense that they 
abstract out the concrete values of timers. Our approach allows to analyze the 
quantitative aspects of SDL timers as well. Second, the previous approaches are 
incorrect, as far as the timing issues are concerned. More precisely, instead of 
just introducing more behaviours, which is unavoidable when the concrete values 
of timers are abstracted out, they simultaneously remove some of the behaviours 
that are allowed by SDL, which may lead to unsound results (so called “false 
positives”). Some concrete examples are given in Section 3.3. The incorrectness 
of the previous attempts also shows that taking the timing issues of SDL into 
account, when using Spin to model check SDL, is not trivial.

We do not claim that our approach is correct, in the formal sense. Ideally, 
one should prove that the approach is sound (no “false positives” are possible) 
and complete (no “false negatives” are possible). In principle, such a correctness 
result cannot be established, due to the lack of formal semantics, both for SDL 
and Promela, which would be simple enough to carry such correctness proofs. 
However, we give some informal justification of the correctness of our approach.

We clearly separate the qualitative and quantitative aspects of SDL timers. 
This allows to analyze the SDL models that use timers, both in the abstract 
and concrete way. The two methods have their own benefits and drawbacks. 
In the abstract case, if DT Spin decides that some safety property holds then 
the property is true for all values of timers, and is thus time independent. This 
may be a desired feature of a model. On the other hand, proving the time 
independence may come at a price: “false negatives” are possible, in the case a 
property does depend on time. The analysis with the concrete values of timers 
does not lead to “false negatives”, but the price may be a bigger state space that 
must be enumerated by DT Spin.

We put some effort in making DT Spin a “conservative” extension of Spin: 
DT Promela is designed in such a way that standard Spin can be used to model 
check DT Promela programs obtained from the SDL models with abstracted 
timers. This may be useful for those who prefer to use a proven technology, 
instead of our experimental DT Spin.

The paper is organized as follows. In Section 2, we give an overview of Spin 
and DT Spin. Section 3 is devoted to the translation of SDL to DT Promela. The
verification method and its application to the MASCARA protocol is presented in Sections 4 and 5. Finally, we conclude with Section 6.

2 Spin and DT Spin

2.1 Spin and Promela

Spin [5] is a software tool that supports the analysis and verification of concurrent systems. The system descriptions are modelled in a high-level language, called Promela. Its syntax is derived from C, and extended with Dijkstra’s guarded commands and communication primitives from Hoare’s CSP.

In Promela, system components are specified as processes that can interact either by message passing, via channels, or memory sharing, via global variables. The message passing can either be buffered or unbuffered (as in Hoare’s CSP). Concurrency is asynchronous (no assumptions are made on the relative speed of process executions) and modelled by interleaving (in every step only one enabled action is performed).

Given a Promela model as input, Spin generates a C program that performs a verification of the system by enumerating its state space, using a depth-first search algorithm. This way, both safety properties (such as absence of deadlock, unspecified message receptions, invalid end states, and assertions) and liveness properties (such as non-progress cycles and eventual reception of messages) can be checked. The most general way of expressing properties in Spin is via so-called never claims, which are best seen as monitoring processes that run in lock step with the rest of the system. The never claims are, in fact, Büchi Automata, and thus can express arbitrary omega-regular properties. Spin provides an automatic translator from formulae in linear-time temporal logic (LTL) to never claims, so it can be used as a full LTL model checker. In case the system violates a property, the trace of actions leading to an invalid state, or a cycle, is reported. The erroneous trace can be replayed, on the Promela source, by a guided simulation.

To cope with the problem of state space explosion, Spin employs several techniques, such as partial-order reduction, state-vector compression, and bit-state hashing.

2.2 DT Spin and DT Promela

DT Spin [2] is an extension of Spin with discrete time. In the time model used in DT Spin, time is divided into slices indexed by natural numbers that can be seen as readings of a fictitious global digital clock that ticks at the end of each slice. The events happening in the same slice are assigned the same clock value, so the elapsed time between events is measured in ticks. In our model, time passes only if no other action in the system is possible.

Since concurrency is modelled by interleaving, all the events happening in one run of a system are totally ordered and thus two events happening in the same slice are not considered necessarily simultaneous. Instead, they are considered
to be ordered, and their ordering inside one slice is determined by the ordering of the run. The properties that depend only on the ordering of events are called *qualitative* while those depending on the elapsed time between events are called *quantitative*.

In order to capture timing features, standard Promela is extended to DT Promela. A new data type, called *timer*, is introduced. It is used to declare variables that represent discrete-time countdown timers. Three new statements that operate on timers are added: set\((\text{tmr}, \text{val})\) activates the timer \(\text{tmr}\), by assigning the integer value \(\text{val}\) to \(\text{tmr}\), reset\((\text{tmr})\) deactivates \(\text{tmr}\), by setting it to \(-1\), and expire\((\text{tmr})\) tests whether the value of \(\text{tmr}\) is 0. Initially, a timer has value \(-1\).

In fact, the new statements are defined as Promela macros, in a special header file included at the beginning of every DT Promela model:

```c
#define timer short /* a short integer */
#define set(tmr,val) tmr = val
#define reset(tmr) tmr = -1
#define expire(tmr) tmr == 0
```

The new statements allow to model a broad class of timing constraints, and other timed statements can easily be defined as Promela macros, by combining *set*, *reset* and *expire* with the control flow statements offered by Promela. There is yet another operation on timers: the *tick* statement decreases the value of all active timers by 1. It is used internally by DT Spin, at the end of every time slice, and is not available to the user.

DT Spin is fully compatible with Spin, and all features of Spin can be used to analyse discrete-time models. In particular, the partial order reduction algorithm of Spin [7, 9] had to be adapted for timed systems [2]. Besides qualitative properties, a broad range of quantitative properties can be verified using boolean expressions on timer values, in the assertions and LTL formulae.

### 3 Translating SDL to DT Promela

The process of model checking an SDL specification is depicted in Figure 1. An SDL specification is pushed through the pipe of translators *sd12if* and *if2pml*, to obtain a DT Promela program that serves as input to DT Spin or Spin. The result of a negative verification experiment (e.g., an erroneous trace) has to be checked manually against the SDL specification.

*sd12if* translates SDL to the language IF (Intermediate Format, [3]) which is a specification language for timed concurrent systems consisting of a fixed number of communicating automata. IF was designed as an intermediate formalism for connecting several industrial formal description techniques, such as LOTOS and SDL, to a number of verification tools developed in the research community.

*sd12if* is implemented with the help of the SDL/API Interface provided by the ObjectGEODE tool [11]. The current implementation of *sd12if* is able to
translate a substantial subset of SDL. The only essential omissions are dynamically created processes and abstract data types.

In the overall SDL to DT Promela translation, sdl2if resolves the hierarchical aspects of SDL by “flattening” the hierarchy of blocks down to bare processes and resolving appropriately the sources, destinations and priorities of the signals exchanged between the processes. On the way, the SDL procedures are eliminated by inlining (when possible), and the syntactic sugar of SDL (enabling conditions, continuous signals, the special input none) is transformed to more primitive constructs. Moreover, some implicit constructs are made explicit (sender, offspring, parent, discarded signals). Details are given in [3].

On the other hand, if2pml performs the translation of the SDL core language, coping with issues that have more semantical flavor (like preserving the atomicity of the transitions or implementing the discard mechanism). Since IF is intended to be an intermediate language for a variety of high-level specification formalisms, it is quite expressive. As not all of its constructs are needed for representing SDL models, if2pml only translates the subset of IF that is relevant to SDL.

IF and sdl2if were developed at Verimag, Grenoble, while if2pml and DT Spin were developed by the authors at the Eindhoven University of Technology. All the tools were developed in the framework of the VIRES project [12].

In what follows we describe the translation from IF to DT Promela in more detail. The presentation is divided into three parts. In Section 3.1 we describe how the SDL processes (i.e., IF automata) are represented in Promela. In Section 3.2, the DT Promela representation of the SDL/IF timers is given. Finally, in Section 3.3, the abstraction from the concrete values of timers is described.

3.1 IF to Promela: Translating Automata

As in Promela, the IF models are sets of processes that communicate via buffers. This provides an almost one-to-one translation of these concepts. Also, the IF data types can directly be translated to their Promela counterparts (with some minor restrictions on the range types).
The way the SDL/IF automata are represented in Promela is fairly standard, and can be grasped by comparing the IF source given in Fig. 2 with its Promela translation in Fig. 3. A state is represented by a Promela label. All the outgoing transitions are translated to the branches of the choice statement associated with the label. The discard mechanism is implemented via self-looping to the same state, after reading a signal that has to be discarded in the state. The atomicity of SDL/IF transitions is preserved by putting the if statement inside the atomic statement.¹

```
process proc: buffer buf;
state
    state1 discard sig3, sig4 in buf
    state2
    ...
transition
    from state1 input sig1 from buf do body1 to state2;
    from state1 input sig2 from buf do body2 to state3;
    ...
```

Fig. 2. A skeleton of an IF program.

```
proctype proc() {
    state1: atomic {
        if
            :: buf?sig1 -> translated_body1; goto state2;
            :: buf?sig2 -> translated_body2; goto state3;
            ...
            :: buf?sig3 -> goto state1; /* discard */
            :: buf?sig4 -> goto state1; /* discard */
        fi
    }

    state2: atomic{
        ...
    }
    ...
}
```

Fig. 3. Promela translation of the structure from Fig. 2

The implementation of if2pml is still in a prototype stage and some SDL features are not supported yet. The most notable omissions are the mechanism of

¹ A special care is taken to correctly handle the stable and non-stable states in IF.
saving a signal, the dynamic creation of processes, and the abstract data types. The implementation of the save mechanism in Promela is possible, but rather involved. It may lead to a substantial growth of the state space during model checking, and for this reason we have chosen to omit it. The dynamic process creation is a basic feature of Promela, so it can easily be implemented once sdl2if supports it. Finding a satisfactory solution for the abstract data types remains a future work for both sdl2if and if2pml.

3.2 IF to DT Promela: Translating Timers

The crucial issue about time in SDL is the detection of timer expirations (time-outs). In SDL, a timer expiration results in sending a timeout pseudo-signal to the input queue of the process the timer belongs to. The timeout signal is then handled as an ordinary signal: it is either consumed, by a transition that is guarded by the timeout signal, or discarded, in a state with no such transitions. In both cases, the corresponding timer is deactivated.

Our implementation of timers does not use such timeout signals. Instead, we associate with each SDL timer a DT Promela timer variable, and detect the expiration of the timer by testing whether the timer variable has value 0, and the timer is deactivated by setting the timer variable’s value to $-1$.

More precisely, for each timer $tmr$ declared in an SDL process, we add a new branch to all the choice statements associated with states (see figure 3). Assume a state, say $state1$, with an outgoing transition guarded by $tmr$. For such a state we add the branch

$$:: \text{expire}(tmr) \rightarrow \text{reset}(tmr); \text{translated}\_\text{body1}; \text{goto} \ state2;$$

If $state1$ has no outgoing transitions guarded by $tmr$, we add the branch

$$:: \text{expire}(tmr) \rightarrow \text{reset}(tmr); \text{goto} \ state1; /* \text{discard} */$$

It turns out that under the time semantics given below (as determined by the ObjectGEODE tool [11]), these two approaches to timers are equivalent. However, the “variable” approach has an advantage over the “signal” approach, from the verification point of view, since it generates smaller state spaces. In the “signal” approach, an additional Promela process (or even several processes) would be needed, in order to generate the timeout signals in a right way. This, together with the overhead of exchanging timeout signals, increases the state space.

In what follows, we give an informal justification of the above mentioned equivalence.

The semantics of time in SDL. Transitions are instantaneous. Time can only progress if at least one timer is active and all SDL processes are waiting for further input signals (i.e., all input queues are empty, except for saved signals). Time progression amounts to performing a special transition that makes
time increment until an active timer expires. In the sequel, we refer to the segments of time separated by the special transition as time slices. (Note that time progression is discretized.)

With each timer there is associated a pseudo-signal and an implicit transition, called a timeout transition. When a timer expires, in some time slice, its timeout transition becomes enabled and can be executed at any point of the time slice. The execution of this transition adds the associated pseudo-signal to the process queue. The timeout transitions of the timers that expire simultaneously can be executed in any order.

If the set or reset operation is performed on a timer after its timeout transition becomes enabled, the timeout transition is disabled. If the timer is set or reset after adding its associated pseudo-signal to the process queue, the pseudo-signal is removed from the queue.

**Model equivalence.** In order to justify the equivalence of the two models we need to show that any execution sequence in the signal model can be simulated by an execution sequence in the variable model, and vice versa. In what follows, we assume that SDL timers are never set to the special value now (i.e., our timer variables are never set to 0, explicitly), and we only concentrate on the simulation of the transitions which relate to timers, since the two models coincide on the untimed features. There are two issues which have to be considered: the set and reset operations on timers, and the expiration of timers.

The set and reset operations coincide in the two models, so this issue does not cause problems. As far as the expiration of timers is concerned, it should be obvious that the time slice in which a timer expires is recognized in the same way, in both models.

The only problematic issue is whether consuming/discard the timeout signals, in the signal model, is properly simulated with our count-down timers, and vice versa. Our claim that, in fact, this is the case is based on the assumption that the following, more direct, translation of SDL/IF to Promela would be correct.

Assume $ts_T$ denotes the timeout signal corresponding to timer $T$, in the signal model. In order to handle the consumption of $ts_T$, like the consumption of an ordinary signal, by an SDL/IF transition guarded by $ts_T$, the following branch

$$::\ buf?ts_T \rightarrow \text{translated\_body1}; \text{goto\ state2};$$

could be added to the Promela translation in figure 3.

Similarly, in order to handle the discarding of $ts_T$ as an ordinary signal, the following branch

$$::\ buf?ts_T \rightarrow \text{goto\ state1}; /\!*\text{discard}\ */$$

could be added to the choice statements that corresponds to the states with no outgoing transitions guarded by $ts_T$.

Observe that the
:: expire(tmr) -> reset(tmr); ...

branches, in our real Promela code, correspond directly to the above branches. More precisely, \texttt{expire(tmr) -> reset(tmr)} corresponds directly to \texttt{buf?ts.T}. Namely, \texttt{expire(tmr)} corresponds to the check whether \texttt{ts.T} is in the input queue, and \texttt{reset(tmr)} corresponds to the removal of \texttt{ts.T} from the queue.

### 3.3 Timer Abstraction

It turns out that standard Spin can also be used to model check a DT Promela model, for a property that does not depend on time. The advantage of using Spin instead of DT Spin is that Spin usually consumes less resources than DT Spin. In order to convert a DT Promela model into a Promela model, it suffices to change the special header file included at the beginning of every DT Promela model (see Section 2.2):

\begin{verbatim}
#define timer bool
#define set(tmr,val) tmr = true
#define reset(tmr) tmr = false
#define expire(tmr) tmr == true
\end{verbatim}

The new definitions abstract out the concrete values of active timers, by consistently mapping them to \texttt{true}. The concrete value \texttt{-1}, which is used to mark an inactive timer, is consistently mapped to \texttt{false} (under the assumption that each timer is initialised to \texttt{false}). Obviously, such an abstraction is sound since no behaviours are lost. More precisely, any behaviour with concrete timers can be simulated with abstract timers, since the active timers are allowed to expire nondeterministically, at any time.

This is not the case in the related approaches [6, 10], where some heuristic approximations of the timer behaviour are used rather than a proper abstraction. The approximations do not only add some behaviours, but also lose some of them, as shown in the following examples.

In [6] the authors try to minimize the number of timer expirations, in the abstract system, that do not correspond to expirations in the concrete system. They take advantage of the observation that, in practice, the transitions guarded by a timer expiration are supposed to resolve the kind of deadlocks when no other transitions except the ones triggered by timeout signals would be enabled. The timers are represented by processes that send the corresponding timeout signals when the special Promela statement \texttt{timeout} becomes enabled. The \texttt{timeout} statement becomes enabled when no other transition is enabled, in the system.

However, there are situations when a behaviour of the concrete system cannot be simulated by the approximate one. Let us consider two concurrent processes, say $P_1$ and $P_2$, specified by the following transition systems:

\[
\begin{align*}
P_1 : & \quad T_1/A \rightarrow B/ \rightarrow \bullet \\
P_2 : & \quad T_2/B \rightarrow A/ \rightarrow \bullet
\end{align*}
\]
where $T_1$ and $T_2$ are timer signals, $A$ and $B$ are normal signals, and $X/Y$ denotes “receive $X$ and send $Y$ to the other process”.

If both $T_1$ and $T_2$ expire simultaneously then, in the concrete system, $P_1$ and $P_2$ may exchange signals $A$ and $B$, and both reach their final states. However, in the abstract system, one of the processes will not be able to reach its final state. Initially, the first transitions of both processes are enabled. If $P_1$ performs its first transition (and thus sends $A$ to $P_2$), the first transition of $P_2$ becomes disabled. $P_2$ must discard $A$, before its first transition becomes enabled again, and thus it will not be able to pass the $A/\$ guard. Similarly, if $P_2$ performs its first transition, $P_1$ will not be able to pass the $B/\$ guard.

In the approximation used in [10], one pseudo-process is used to handle all timer expirations. After each set operation this pseudo-process immediately sends the corresponding timeout signal to the input queue of the process that sets the timer. As a result, if an SDL process uses two timers, they can only expire in the order they are set, no matter what values they are set to. Obviously, the behaviour in which an SDL process sets the timers $T_1$ and $T_2$ (in that order) to such values that $T_2$ expires before $T_1$ cannot be simulated by the abstract system.

4 Verification Methodology

The aim of this section is to present the way the real verification process is performed. Industrial-size SDL models normally cannot be verified with existing model-checking tools as a whole, so the first natural task is to split a protocol into some relatively autonomous parts of a reasonable size and apply to them compositional verification. Fortunately, due to their block-structure, SDL-systems can be usually split in a natural way without much effort.

The obtained sub-models are not self-contained, i.e. the behaviour of their environment is not specified. Since Spin can only model-check closed systems we need to close our model first. We achieve this by adding an “environment” process specified in SDL at the same hierarchy level as the extracted model itself.

The simplest possible approach is to construct an environment producing all the “possible” behaviours but practice shows that this is not of much use in real life. Such an environment leads to adding to the model too many erroneous behaviours and thus one gets too many “false negatives” during model-checking. Local livelocks, cycles with non-progressing time, and non-existing deadlocks are the typical examples of those false-errors. Moreover, since many redundant behaviours are added, this may also lead to a state explosion. Another possibility is to construct an environment being able to send/receive a signal whenever the modelled system is ready to get/send it. Applying such an approach reduces the added behaviours but it still adds some unwanted behaviours caused by sending non-realistic signal sequences.

Both these approaches are not safe: in case of adding non-progressing time cycles, we lose some behaviour of the system. So they can be considered as
a kind of heuristics only, that may be of some use at the first stage of system debugging.

A different approach is to provide an SDL-specification of the “right” environment, i.e. the one, which faithfully models the assumptions under which the component was designed, giving an abstraction of a real environment. Although it makes the soundness of verification results dependent on the quality of the environment model, it usually turns out to be a practical method.

The closed SDL-model can be automatically translated into DT Promela through the translators sdl2if and if2pml. Then one should choose between verification of the concrete model with DT Spin and verification of the model with abstracted time in the standard Spin. First, the built-in Spin features (finding deadlocks and livelocks, unreachable code) are used. After correcting discovered structural errors, functional properties defined by a designer or drawn from the informal specification of the system can be verified.

It would seem obvious to verify all non-timed properties with an abstracted-time model and all timed properties with a concrete model. However, sometimes it is more convenient to verify non-timed properties with a concrete model as well. If some functional property was proved with the abstracted-time model, it is proved for all possible values of timers. However if it was disproved, or a deadlock in the model was found, the next step is to check whether the erroneous trace given by Spin is a real error in the system or it is a false error caused by adding erroneous behaviour either with abstracting from time or with too abstract specification of the environment. It can happen that the property does not hold for the concrete model, however the erroneous trace given by Spin is one of the added behaviours. This behaviour cannot be reproduced for the SDL model with SDL-simulation tools and we cannot conclude whether the property holds or not.

One can not force Spin to give the trace from the non-added behaviours. DT Spin allows to reduce the set of added behaviours guaranteeing that timers are expiring in the correct order. In our verification experiments we had a number of cases when application of DT Spin, instead of Spin, gave a chance to get a real erroneous trace and disprove the property (see the next section).

5 Case Study: Verifying the MASCARA Protocol

We have applied our tools to the verification of an industrial-size communication protocol called MASCARA (Mobile Access Scheme based on Contention and Reservation for ATM), developed by the WAND (Wireless ATM Network Demonstrator) consortium [13]. The protocol is an extension of the ATM (Asynchronous Transfer Mode) networking protocol to wireless networks.

A wireless ATM network comprises of a fixed wire-based ATM network that links a number of geographically distributed access points, which transparently extend ATM services to mobile users, via radio links operating in the 5.2 GHz frequency band and achieving data rates of 23.5 MBits/s. The task of MASCARA is to mediate between the mobile users and the access points, to offer
the mobile users the advantages of the standard wire-based ATM protocol: high reliability, capacity-on-demand, and service-independent transport. The transparent extension of ATM over radio links is a challenge, as standard ATM has not been designed to work in a wireless environment. The radio medium is characterised by a high bit error rate, the transmission mode is typically broadcast (or at least multicast) and the scarcity of available radio bandwidth calls for a time division duplex (i.e. half duplex) mode. All this leads to the necessity of extra functionality allowing to cope with these dissimilar environmental properties.

A crucial feature of MASCARA is the support of mobility. A mobile terminal (MT) can reliably communicate with an access point (AP) only within some area called the AP’s cell. Whenever an MT moves outside the cell of its current AP it has to perform a so-called handover to an AP whose cell the MT has moved into. A handover must be managed transparently with respect to the ATM layer, maintaining the agreed quality of service for the current connections. So the protocol has to detect the need for a handover, select a candidate AP to switch to and redirect the traffic with minimal interruption.

The protocol is too large to be automatically verified as a whole so we have concentrated our efforts on the verification of a substantial part called MCL (MASCARA Control). The main purpose of MCL is to support the mobility issues. It contains 9 processes, each with up to 10 states (6 on average). Its main function is to monitor the current radio link quality, gather information about radio link qualities of its neighbouring APs (to make a quick handover, in the case of deterioration of the current link quality), and switch from one AP to another (during the handover procedure).

During the first phase of the verification, several deadlocks were found. Most of them were related to improper synchronisation between various request/confirm subprotocols (a component requests a service from another component and waits until the confirmation arrives that the request is either rejected or served). The second source of deadlocks was the potential race conditions between various components of MCL, due to the fully asynchronous communication in SDL (non-blocking sending of messages). The following example describes one of the race conditions we have found.

Most MASCARA components must be initialised before they are prepared to serve requests from other components. The components are grouped into various tree-like hierarchies, and the initialisation phase for a group is triggered by sending the INIT signal to the group’s root. Each node that receives the INIT signal resends the signal to all its children. However, in such a cascade of INIT signals there is a possible race condition: Spin found a trace in MCL where an already initialised component tried to immediately request a service from a component that had not been initialised yet. Such a request was simply discarded and thus its confirmation was never sent back, leading to a deadlock in MCL.

After correcting these errors, we continued the verification, by exploiting another useful Spin feature—unreachable code diagnosis. The analysis of the unreachable code reported by Spin revealed that some code for serving one par-
ticular request is never reached and thus the request for that particular service was never confirmed. Further analysis showed that there was a local deadlock possible. (This local deadlock could not be discovered by Spin, in previous experiments, since Spin can only discover global deadlocks.)

Finally, we verified some functional properties that we derived from the informal specification. An example of such a property is given below.

One of the tasks of MCL is to periodically update a list that contains information about the quality of radio links with the neighbouring APs. This updating phase is called the TI procedure. In order to judge the quality of a radio link with a particular AP, MCL must connect to that particular AP, so the connection with the current AP is suspended during the whole TI procedure. Therefore, another procedure, checking the quality of the current connection and making the decision on the necessity of a handover, should not interfere with TI procedure.

This requirement was encoded as a simple observer for the following safety property: a handover request never comes in between the two messages that mark the beginning and the end of the TI procedure. The verification experiment with Spin, which was supposed to confirm this property, showed instead a trace violating it. Unfortunately, the erroneous trace could not be simulated by the SDL model of MCL (so we got a so-called “false negative”). However, when exactly the same experiment was repeated with DT Spin we got a different erroneous trace which could then be simulated by the SDL model. Thus, the property was indeed violated, DT Spin allowed us to find yet another error in MASCARA protocol.

The explanation of the different results obtained with Spin and DT Spin is obvious. The TI procedure is triggered by a timer, so the behaviour of the TI protocol could indeed depend on proper timing. In the model with abstracted time, as used in Spin, timers can expire at any time, so Spin can produce a wrongly timed trace that is not accepted by an SDL simulator (which allows only properly timed traces, of course). After finding a way to re-design the components, some other functional properties were proved.

When we planned the first series of verification experiments we expected to reach the limits of Spin and DT Spin quickly. We were proved wrong. In the experiment that consumed the most resources, Spin reported the following statistics:

State-vector 416 byte, depth reached 3450, errors: 0
55959 states, stored
23.727 memory usage for states (in MB)
25.582 total memory usage (in MB)
14.2 total time (in seconds)

which should be read in the following way:

State-vector 416 byte is the memory needed to represent one state.
55959 states, stored is the number of different states found in the model (all the states must be kept during the state space enumeration, so 23.727MB memory was needed for states).
depth reached 3450 is the greatest depth reached during the depth-first search (since Spin must keep a state stack of at least this depth, about 1.7MB was needed in addition to the state memory).

It is quite likely that with our 2048MB memory we will be able to handle even more complex case studies.

6 Conclusions and future work

We have developed a tool, if2pml, that enables the translation from SDL to Promela. It can be used, together with the accompanying tool sdl2if, to model check SDL specifications with Spin.

Our approach preserves the timing aspects of SDL. This is in contrast to other translators that we know, which only approximate timing aspects. SDL timers, which expire by sending signals, are in our approach translated into the timer variables provided by DT Promela. We have argued the correctness of this translation.

The approach has been successfully used on an industrial case study. More information is available from http://radon.ics.ele.tue.nl/~vires/public/results.

As a future work, we consider to extend the tool by implementing the translation of dynamic process creation in SDL and the save construct. SDL supports various styles of specifying data types. It needs to be investigated how the specification of data aspects can be combined with the translation from SDL to Promela.

Acknowledgments

We gratefully acknowledge VERILOG for giving us free access to their Object-GEODE tool. We would also like to thank Marius Bozga, Lucian Ghirvu, Suzanne Graf (Verimag, Grenoble) and Gerard Holzmann (Bell Labs) for fruitful discussions during the implementation of the if2pml tool.

References


6. A Heuristic for Symmetry Reductions with Scalarsets

Chapter 6 is a combination of revised versions of the papers:


and

A Heuristic for Symmetry Reductions with Scalarsets

Dragan Bošnački\textsuperscript{1}, Dennis Dams\textsuperscript{2}, and Leszek Holenderski\textsuperscript{1}

\textsuperscript{1} Dept. of Computing Sci., Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
\textsuperscript{2} Dept. of Electrical Eng., Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
\{D.Bosnacki,D.Dams,L.Holenderski\}@tue.nl

Abstract. We present four versions of a new heuristic for coping with the problem of finding (canonical) representatives of symmetry equivalence classes (the so-called orbit problem), in symmetry techniques for model checking. The practical implementation of such techniques hinges on appropriate workarounds of this hard problem, which is equivalent to graph isomorphism. We implemented the four strategies on top of the Spin model checker as a symmetry-reduction package called SymmSpin. Using SymmSpin we compared the performance of the strategies on several examples, with encouraging results.

1 Introduction

One way to combat the state explosion problem in model checking \cite{5,8} is to exploit symmetries in a system description. In order to grasp the idea of symmetry reduction, consider a mutual exclusion protocol based on semaphores. The (im)possibility for processes to enter their critical sections will be similar regardless of their identities, since process identities (pids) play no role in the semaphore mechanism. More formally, the system state remains behaviorally equivalent under permutations of pids. During state-space exploration, when a state is visited that is the same, up to a permutation of pids, as some state that has already been visited, the search can be pruned. The notion of behavioral equivalence used (bisimilarity, trace equivalence, sensitivity to deadlock, fairness, etc.) and the class of permutations allowed (full, rotational, mirror, etc.) may vary, leading to a spectrum of symmetry techniques.

The two main questions in practical applications of symmetry techniques are how to find symmetries in a system description, and how to detect, during state-space exploration, that two states are equivalent. To start with the first issue: as in any other state-space reduction method based on behavioral equivalences, the problem of deciding equivalence of states requires, in general, the construction

\footnote{This research has been supported by the VIRES project (Verifying Industrial Reactive Systems, Esprit Long Term Research Project #23498).}
of the full state space. Doing this would obviously invalidate the approach, as it is precisely what we are trying to avoid. Therefore, most approaches proceed by listing sufficient conditions that can be statically checked on the system description. The second problem, of detecting equivalence of states, involves the search for a canonical state by permuting the values of certain, symmetric, data structures. In [4] it was shown that this orbit problem is at least as hard as testing for graph isomorphism, for which currently no polynomial algorithms are known. Furthermore, this operation must be performed for every state encountered during the exploration. For these reasons, it is of great practical importance to work around the orbit problem. In practice, heuristics for the graph isomorphism problem can be reused to obtain significant speedups. In case these do not work, one can revert to a suboptimal approach in which (not necessarily unique) normalized states are stored and compared.

The use of symmetry has been studied in the context of various automated verification techniques. We mention here only a few papers that are most closely related to our work, which is in the context of asynchronous systems. For a more complete overview we refer to the bibliography of [20]. Emerson and Sistla have applied the idea to CTL model checking in [10], with extensions to fairness in [13] and [16]. In [11], Emerson and Trefler extended the concepts to real-time logics, while in [12] they considered systems that are almost symmetric. Clarke, Enders, Filkorn, and Jha used symmetries in the context of symbolic model checking in [4]. Emerson, Jha, and Peled, and more recently Godefroid, have studied the combination of partial order and symmetry reductions, see [9,15]. Our work draws upon the ideas of Dill and Ip [18–20]. They introduce, in the protocol description language Murϕ, a new data type called scalarset by which the user can point out (full) symmetries to the verification tool. The values of scalarsets are finite in number, unordered, and allow only a restricted number of operations, that do not break symmetry; any violations can be detected at compile time.

We take the approach of Ip and Dill as the starting point. In order to work around the orbit problem, we follow their idea of splitting the state vector — the following is adapted from [18]:

We separate the state into two parts. The leftmost (most significant) part is canonicalized (by picking the lexicographically smallest equivalent as a representative). Because the same lexicographical value may be obtained from different permutations, we may have a few canonicalizing permutations for this part of the state. The second, rightmost part is normalized by one of the permutations used to canonicalize the first part. The result is a normalized state of a small lexicographical value.

In this paper, we improve on this idea by exploiting another degree of freedom, namely the freedom to choose the ordering of variables in the state vector, on which the lexicographical ordering is based. Viewed differently, we reschedule the positions of variables in the state vector — but only conceptually so! — before splitting it. In doing so, the goal is to move certain variables to the left so as to reduce the number of permutations that is determined by canonicalizing
the leftmost part. Reshuffling of the state vector is done by searching for an array that is indexed by a scalarset type. This main array is then conceptually positioned at the leftmost end of the state vector. This results in a new heuristic for normalization, called the sorted strategy. A second improvement ensues by not using one of the permutations obtained from canonicalizing the leftmost part of the state vector, but using all of them in order to canonicalize the second part. This segmented strategy induces the same reduction as canonicalization of the state vector without splitting—which we have also implemented for reference purposes (full strategy)—but involves a much smaller overhead in time, as is demonstrated by our experiments.

We have also implemented a variation on this, which is particularly useful when no main array occurs in the system description. Namely, in the case that the process identities are of type scalarset, a main array can be coined by putting the program counters of the individual processes together in an array. The resulting strategies are called pc-sorted and pc-segmented.

In order to compare these 5 strategies, we have implemented them on top of the Spin model checker [17]. The implementation resulted in a symmetry-reduction package called SymmSpin. An overview of SymmSpin is given in Section 4.

Building upon results reported in [9], our extension is compatible with Spin’s partial order reduction algorithm, which is another, orthogonal approach to reduce the state space, and indeed one of the major strengths of the Spin tool.

We are aware of only one other attempt to extend Spin with symmetry reductions [23]. With their implementation the user himself (herself) has to write a function that computes the normalized state. As a consequence, it is hard to see how this approach can be generalized. Moreover, it might require quite a deep knowledge of Spin’s internal workings.

Our current implementation preserves all safety properties. A large number of experiments have been conducted that show good results (Section 5).

2 Preliminaries

A transition system is a tuple $T = (\mathcal{S}, s_0, \rightarrow)$ where $\mathcal{S}$ is a set of states, $s_0 \in \mathcal{S}$ is an initial state, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation. We assume that $\mathcal{S}$ contains an error state $e \neq s_0$ which is a sink state (whenever $e \rightarrow s$ then $s = e$).

An equivalence relation on $\mathcal{S}$, say $\sim$, is called a congruence on $T$ iff for all $s_1, s_2, s'_1 \in \mathcal{S}$ such that $s_1 \sim s_2$ and $s_1 \rightarrow s'_1$, there exists $s'_2 \in \mathcal{S}$ such that $s'_1 \sim s'_2$ and $s_2 \rightarrow s'_2$. Any congruence on $T$ induces a quotient transition system $T/\sim = (\mathcal{S}/\sim, [s_0], \Rightarrow)$ such that $[s] \Rightarrow [s']$ iff there exists a transition $q \rightarrow q'$, such that $q \in [s], q' \in [s']$.

A bijection $h : \mathcal{S} \rightarrow \mathcal{S}$ is said to be a symmetry of $T$ iff $h(s_0) = s_0$, $h(e) = e$, and for any $s, s' \in \mathcal{S}$, $s \rightarrow s'$ iff $h(s) \rightarrow h(s')$. The set of all symmetries of $T$ forms a group (with function composition).
Any set \( A \) of symmetries generates a subgroup \( G(A) \) called a symmetry group (induced by \( A \)). \( G(A) \) induces an equivalence relation \( \sim_A \) on states, defined as

\[ s \sim_A s' \text{ iff } h(s) = s', \text{ for some } h \in G(A) \]

Such an equivalence relation is called a symmetry relation of \( T \) (induced by \( A \)). The equivalence class of \( s \) is called the orbit of \( s \), and is denoted by \( [s]_A \).

Any symmetry relation of \( T \) is a congruence on \( T \) (Theorem 1 in [19]), and thus induces the quotient transition system \( T/\sim_A \). Moreover, \( s \) is reachable from \( s_0 \) if and only if \( [s]_A \) is reachable from \( [s_0]_A \) (Theorem 2 in [19]). This allows to reduce the verification of safety properties of \( T \) to the reachability of the error state \([e]\) in \( T/\sim_A \) (via observers, for example).

In principle, it is not difficult to extend an enumerative model checker to handle symmetries, i.e., to explore \( T/\sim_A \) instead of \( T \). Provided with a set \( A \) of symmetries and a function \( \text{rep} : S \to S \) which for a given state \( s \) picks up a representative state from \( [s]_A \), one can explore \( T/\sim_A \) by simply exploring a part of \( T \), using \( \text{rep}(s) \) instead of \( s \). A generic algorithm of this type is given in Section 4. In the sequel, by a reduction strategy we mean any concrete \( \text{rep} \).

When choosing a reduction strategy one has to deal with two contradictory requirements: \( \text{rep} \) should both lead to a substantial reduction of the explored state space and be computationally inexpensive. Obviously, the best reduction of the state space is obtained if \( \text{rep} \) is canonical (i.e., \( \text{rep}(s) = \text{rep}(s') \) whenever \( s \sim_A s' \)) since then the explored part of \( T \) is isomorphic with \( T/\sim_A \). On the other hand, a canonical \( \text{rep} \) may be computationally costly and the time overhead caused by using it can substantially exceed the time savings gained by exploring a smaller state space.

In Section 3 we formally define several reduction strategies and then, in Section 5, compare their performance on several examples.

### 3 Reduction Strategies

Assume a model is specified by a program in a programming language based on shared variables. The variables are typed, and the types are of two kinds only: simple types (say, finite ranges of integers) and array types. An array type can be represented by a pair \( I \mapsto E \) of types where \( I \) is the type of indices and \( E \) is the type of elements. \( I \) must be simple while \( E \) can be any type that does not depend on \( I \mapsto E \). Let \( D_T \) denote the set of all values of type \( T \) then a value of an array type \( I \mapsto E \) is a function \( a : D_I \to D_E \). We write \( a[i] \) instead of \( a(i) \).

We assume that a program specifies a set of processes run in parallel, and that the processes use only global variables. (The latter assumption is not essential. We use it only to simplify the presentation.) Let \( V \) denote the set of variables used in the program and \( D \) denote the union of \( D_T \), for all types \( T \) used in the program. A program \( P \) induces a transition system \( T_P \), via a formal semantics of the programming language. We assume that in the semantics states are pairs \((s_V, s_{PC})\) where \( s_V : V \to D \) is the valuation of program variables
and \( s_{PC} \) represents the values of program counters, for each process in \( P \). For \( s = (s_V, s_{PC}) \) and \( v \in V \), \( s(v) \) means \( s_v(v) \).

In order to simplify the detection of symmetries in \( T_P \), we assume a special simple type called \( \text{Pid} \) which is a scalar set (i.e., an unordered subrange of integers from 0 to \( n-1 \), for some fixed \( n \)). Its elements are called \textit{pids} (for process identifiers). We assume that all programs are of the form \( B||C_0||\cdots||C_{n-1} \) where \( B \) is a base process and \( C_0, \ldots, C_{n-1} \) are instances of a parameterized family \( C = \lambda i : \text{Pid} . C_i \) of processes. Notice that \( \text{Pid} \) can be used as any other simple type, for example, to index arrays.

For such a class of programs, it is convenient to treat the program counters of the family \( C \) as an array indexed by pids, and to consider it as a global variable. Thus, in \( s = (s_V, s_{PC}) \), the program counters of the family \( C \) are in fact “stored” in the component \( s_V \), and \( s_{PC} \) represents only the program counter of \( B \).

Let \( \mathcal{P} \) denote the set (in fact, the group) of all pid permutations. A pid permutation \( p : \text{Pid} \rightarrow \text{Pid} \) can be lifted to states, via \( p^* : S \rightarrow S \). Intuitively, \( p^* \) applies \( p \) to all values of type \( \text{Pid} \) that occur in a state, including the pids used as array indices. Formally, for \( s = (s_V, s_{PC}) \), we have \( p^*(s) = (\lambda v. \pi_{\text{type}(v)}(s_v(v)), s_{PC}) \) where \( \pi_T : D_T \rightarrow D_T \) is defined as

\[
\pi_T(d) = d \quad \text{if } T \text{ is a simple type other than } \text{Pid}
\]

\[
\pi_{\text{Pid}}(d) = p(d)
\]

\[
\pi_{\text{E}(a)}(d) = \lambda i \in \mathcal{D}_T. \pi_{\text{E}}(a(\pi_T(i)))
\]

In fact, \( p^* \) is a symmetry of \( T_P \) (Theorem 3 in [19]). Moreover, \( \mathcal{P}^* = \{p^* : p \in \mathcal{P} \} \) is a symmetry group of \( T_P \) (since \( (\cdot)^* \) preserves the group operations of \( \mathcal{P} \)).

All the reduction strategies considered in this paper have the same pattern. For a given state, a set of pid permutations is generated, and each permutation is applied to the given state. All the states obtained in this way are equivalent w.r.t. the symmetry relation, and we choose one of them as a representative. Formally, this process is specified by a function \( \text{rep} : S \rightarrow S \) defined as

\[
\text{rep}(s) = \mu(\{p^*(s) : p \in \pi(s)\})
\]

where \( \mu : 2^S \rightarrow S \) is a choice function (i.e., \( \mu(X) \in X \), for any nonempty \( X \subseteq S \)) and \( \pi : S \rightarrow 2^P \) generates a set of pid permutations for a given state.

We call such \( \text{rep} \) a general reduction strategy since it is parameterized by \( \pi \) and \( \mu \). A concrete reduction strategy is obtained from \( \text{rep} \) by fixing some \( \pi \), and is denoted by \( \text{rep}_\pi \). In the sequel, we consider several concrete strategies which we call full, sorted, segmented, pc-sorted and pc-segmented. The names denote the respective \( \pi \) functions in \( \text{rep}_\pi \), and whenever we say “strategy \( \pi \)” we really mean “strategy \( \text{rep}_\pi \)”.

In the full strategy, all pid permutations are taken into account, thus \( \text{full}(s) = \mathcal{P} \). Since this strategy is canonical, for any choice function \( \mu \), it leads to the best reduction of the state space. However, it is computationally intensive.

In order to improve the full strategy we make two assumptions. In fact, the assumptions are needed only for presentation purposes (we want to derive the
improvements to full, and not just state them). As it will turn out later, they can be dropped.

First, we assume a choice function of a particular kind, namely the one which picks up the lexicographically smallest state, under some lexicographical ordering of states. Formally, let us assume that each simple type used in a program is totally ordered, and that the order of pids is just the natural order of the set \{0, \ldots, n - 1\}.\footnote{This is not inconsistent with the assumption that Pid is a scalarset, and thus unordered. In fact, a scalarset can be ordered but a program that uses such a scalarset must not depend on the order (all the models of the program obtained under different orderings must be isomorphic).} As usual, such a total order can be lifted to the lexicographical order on arrays (by considering them as vectors), and then to states (by considering them as vectors).

Second, we assume that program \(P\) uses a variable \(M : \text{Pid} \rightarrow E\) which is an array indexed by pids and whose elements do not involve pids. \(M\) is called a main array. Most real protocols specified in a language based on shared variables, and of the form \(B||C_0||\cdots||C_{n-1}\), use such an array, either directly or indirectly. (Notice that each local variable declared in the parametric program \(C = \lambda i : \text{Pid}. C_i\), and whose type does not involve Pid, can be considered an element of an array indexed by Pid, when lifted to the global level.) We further assume that \(M\) dominates in the total order used by our particular choice function \(\mu\), in the sense that the main array is located at the beginning of the state vector. (If it is not the case, the state vector can be reshuffled.)

Let us consider a state \(s\). Notice that if \(\text{rep}_\text{full}(s) = s'\) then \(s'(M)\) must be sorted, w.r.t. the ordering of \(E\), due to our particular choice of \(\mu\). So instead of considering all pid permutations in full it is enough to consider only those which sort \(s(M)\). (Notice that there may be more than one sorting permutation, if \(s(M)\) contains repeated values.) Let \(\tilde{s}(M)\) denote the set of all sorting permutations.

In the \textit{sorted} strategy we consider just one pid permutation \(\tilde{p} \in \tilde{s}(M)\) obtained by applying a particular sorting algorithm to \(s(M)\). (Formally, \(\text{sorted}(s) = \{\tilde{p}\}\).) Obviously, the \textit{rep}_{\text{sorted}} strategy is faster then \textit{rep}_{\text{full}}. Unfortunately, it is not canonical since \(\tilde{p}\) minimizes only \(s(M)\) and not necessarily the whole \(s\). (If \(s(M)\) contains repeated values then there may be another sorting permutation \(p\) such that \(p^*(s)\) is smaller than \(\tilde{p}^*(s)\).)

In the \textit{segmented} strategy we consider all the permutations in \(\tilde{s}(M)\).\footnote{The name “segmented” comes from the way we have implemented the computation of \(\text{segmented}(s)\). We first sort \(s(M)\), locate all the segments of equal values in the sorted array, and permute these segments independently.} (Formally, \(\text{segmented}(s) = \tilde{s}(M)\).) Obviously, \(\text{rep}_{\text{segmented}}\) is canonical, for the particular choice function we have assumed. Moreover, \(\text{rep}_{\text{segmented}}(s) = \text{rep}_{\text{full}}(s)\), for any \(s\).

As an example, consider the following picture showing a state vector before and after sorting its main array \(M\).
an implicit array indexed by pids, namely the array of program counters for processes $C_0, \ldots, C_{n-1}$. Thus, we can consider the variants of \texttt{sorted} and \texttt{segmented} in which we use the array of program counters instead of $M$. The variants are called \texttt{pc-sorted} and \texttt{pc-segmented}, respectively. If $P$ contains a main array as well then both \texttt{sorted/segmented} and \texttt{pc-sorted/pc-segmented} are applicable to $P$, so the question arises which of the combinations is better. We cannot say much about \texttt{sorted} versus \texttt{pc-sorted}, in general. However, \texttt{segmented} and \texttt{pc-segmented} are both canonical, so they are equally good, as far as the reduction of a state space is considered.

The following result allows us to drop the assumption about our particular $\mu$ function.

\textbf{Theorem 1.} For any choice function $\mu$, $\text{rep}_{\text{segmented}}$ is canonical.

\textit{Proof.} Let $P(s) = \{p^*(s) : p \in \text{segmented}(s)\}$. We have to show that if $s \sim s'$, i.e., there exists a pid permutation $p$ such that $p^*(s) = s'$, then $\mu(P(s)) = \mu(P(s'))$. In order to show it for any choice function $\mu$, it is enough to show that $P(s) = P(s')$. Assume $s_1 \in P(s)$ then $s_1 = p_1^*(s)$ for some pid permutation $p_1$, where $p_1^*$ sorts the array $M$ in $s$. Observe that $s = (p^{-1})^*(s')$ where $p^{-1}$ is the inverse of $p$. Hence, $s_1 = p_1^*((p^{-1})^*(s'))$. Since $p_1^* \circ (p^{-1})^*$ sorts the array $M$ in $s'$, $s_1 \in P(s')$. Similarly, if $s_1 \in P(s')$ then $s_1 \in P(s)$. 

}\end{example}
Remark 1. In the proof, the only step specific to the segmented strategy is the observation that if \( p \) establishes the equivalence of states \( s \) and \( s' \), and \( p_1^* \) sorts the array \( M \), then also \( p_1^* \circ (p^{-1})^* \) sorts the array \( M \). Thus, the theorem can be generalized to any strategy \( \pi \) that preserves pid permutations in the following sense: if \( p \) establishes the equivalence of states \( s \) and \( s' \) then for any \( p_1 \in \pi(s) \), \( p_1^* \circ (p^{-1})^* \in \pi(s') \). For example, this condition holds for the full and pc-segmented strategy as well.

The theorem has an important practical consequence. Suppose we want to extend an existing enumerative model checker with symmetry reductions. Usually, a model checker stores a state in a continuous chunk of memory. A very efficient way of implementing \( \mu \) is to choose the lexicographically smallest chunk by simply comparing such memory chunks byte by byte (for example, in C, the built-in function `memcmp` could be used for this purpose). By Theorem 1, such \( \mu \) is a choice function, no matter where the main array resides in the memory chunk. Thus, the requirement that the main array dominates in the total order used by \( \mu \) is not needed. Also, the array of program counters, used in pc-sorted and pc-segmented, need not even be be contiguous. As a consequence, we do not need to reshuffle a state vector in order to use the presented strategies.

Finally, the assumption that programs are of the form \( B \| C_0 \| \cdots \| C_{n-1} \) with process indices in the scalarset \( \text{Pid} \) is only needed to formalize the pc-sorted and pc-segmented strategies. For segmented and sorted, the only essential assumption about \( \text{Pid} \) is that it is a distinguished scalarset.

In Section 5 we compare, on several examples, the performance of all the concrete strategies introduced above.

4 Extending Spin with Symmetry Reductions

In order to compare the performance of the various reduction strategies in practice, we have embedded them into the enumerative model checker Spin [17]. The result of our extension of Spin with a symmetry package is called SymmSpin.

When extending an existing enumerative model checker, in the spirit of [19], two essential problems have to be solved. First, the input language must be extended, to allow the usage of scalarsets in a program that specifies a model. Second, a symmetry reduction strategy must be added to the state space exploration algorithm. Both problems are non-trivial. As far as the extension of the input language is concerned, a compiler should be able to check whether the scalarsets really induce a symmetry relation (i.e., whether a program does not rely on the order of the range of integers that represents a scalarset). As far as adding a reduction strategy is concerned, the \( \text{rep} \) function should be implementable in an efficient way.

In order for a model checker to be a successful tool for the verification of symmetric systems, good solutions are needed for both problems. However, there does not seem to be much sense in putting an effort into solving the language extension problem without having an efficient reduction strategy. That is why in SymmSpin we have mainly concentrated on the second problem.
As far as the first problem is concerned, it could be solved just by lifting the syntactical extensions of Murϕ [19] to Spin. Although not entirely straightforward, it should not pose fundamental difficulties. Unfortunately, to do it in the right way, one would need to extensively modify the existing Promela parser. In SymmSpin, we have tried to avoid this effort, as explained in Section 4.3.

4.1 A Modified State Space Exploration Algorithm

In principle, extending an existing enumerative model checker to handle symmetries is not difficult, once the rep function is implemented. Instead of using a standard algorithm for exploring $T = (\mathcal{S}, s_0, \rightarrow)$, as depicted in Fig. 1, one explores the quotient $T/\sim$ using a simple modification of the algorithm, as depicted in Fig. 2. (This modification is borrowed from [19].)

1 reached := unexpanded := \{s_0\};
2 while unexpanded \neq \emptyset do
3 remove a state s from unexpanded;
4 for each transition s \rightarrow s' do
5 if s' = error then
6 stop and report error;
7 if s' \notin reached then
8 add s' to reached and unexpanded;

Fig. 1. A standard exploration algorithm

1 reached := unexpanded := \{\text{rep}(s_0)\};
2 while unexpanded \neq \emptyset do
3 remove a state s from unexpanded;
4 for each transition s \rightarrow s' do
5 if s' = error then
6 stop and report error;
7 if rep(s') \notin reached then
8 add rep(s') to reached and unexpanded;

Fig. 2. A standard exploration algorithm with symmetry reductions

In practice, we had to overcome several problems, due to idiosyncrasies of Spin. For example, it turned out that the operation “add rep(s') to unexpanded” is difficult to implement reliably, due to a peculiar way Spin represents the set unexpanded as a set of “differences” between states rather than states themselves. For this reason, we had to change the exploration algorithm given in Fig. 2. In our\(^3\) algorithm, the original states, and not their representatives, are used to

\(^3\) We acknowledge a remark from Gerard Holzmann which led us to this algorithm.
generate the state space to be explored, as depicted in Fig. 3. (The algorithms differ only in lines 1 and 8.)

\begin{verbatim}
1 reached := \{rep(s_0)\}; unexpanded := \{s_0\};
2 while \(\text{unexpanded} \neq \emptyset\) do
3    remove a state \(s\) from \(\text{unexpanded}\);
4    for each transition \(s \rightarrow s'\) do
5        if \(s' = \text{error}\) then
6            stop and report error;
7        if \(\text{rep}(s') \notin \text{reached}\) then
8            add \(\text{rep}(s')\) to \(\text{reached}\) and \(s'\) to \(\text{unexpanded}\);
\end{verbatim}

Fig. 3. Our exploration algorithm with symmetry reductions

Using the notation of Section 2, consider a state transition system \(T = (S, s_0, \rightarrow)\) and some group \(G(A)\) of symmetries. Since the symmetry relation \(\sim_A\) is a congruence, obviously the algorithm in Fig. 3 visits at least one representative of each equivalence class (orbit) induced by \(\sim_A\). In other words, after the algorithm ends, provided that no error is reported, for each state \(s \in S\), there is at least one \(s' \in \text{reached}\) such that \(s' \in [s]_A\). Using the results of [19], this implies directly the correctness of the algorithm in Fig. 3, i.e., the algorithm reports an error iff there exists an error state \(e \in S\).

If the \(\text{rep}\) function is canonical, the algorithms from Fig. 2 and 3 explore the same number of states, provided that no error is reported. This follows from the following reasoning. The total number of visited states is bounded by the number of representatives, i.e., the number of different states \(s\) such that \(s = \text{rep}(s')\), for some \(s' \in S\). This is because the number of visited states is equal to the number of states which are removed from \(\text{unexpanded}\) in line 3. Since a state \(s\) is added to \(\text{unexpanded}\) only if its representative is not already in \(\text{reached}\) (lines 7 and 8), the total number of states which are added to \(\text{unexpanded}\) cannot exceed the number of representatives. On the other hand, as we already mentioned, the algorithm is guaranteed to visit at least one representative of each equivalence class. Thus, if \(\text{rep}\) is canonical, the upper and lower bounds of visited states are the same and they are equal to the number of equivalence classes. Moreover, the number of explored transitions is also the same in both algorithms. This follows from the fact that for each state \(s\) for which our algorithm computes its successors with the statement “for each transition \(s \rightarrow s'\) do” (line 4), the algorithm in Fig. 2 computes the successors of \(\text{rep}(s)\). Since \(s\) and \(\text{rep}(s)\) are related by a symmetry, they must have the same number of successors (recall that any symmetry is a bijection).

Under some quite natural constraints on the nondeterministic parts of the algorithms (and provided that \(\text{rep}\) is canonical), one can show that both of them visit the same number of states and explore the same number of transitions, even if an error is reported:
Proposition 1. Given a transition system $T$, let $rep$ be canonical and suppose the following holds for the algorithms in Figures 2 and 3:

1. The order in which a state $s$ is chosen from unexpanded in line 3 (remove a state $s$ from unexpanded) in the algorithm in Fig. 3 is such that: For each two states $s_1, s_2$, it is the case that $s_1$ is chosen before $s_2$ iff in the same line in the algorithm in Fig. 2 $rep(s_1)$ is chosen before $rep(s_2)$.
2. The order in which transitions are explored in line 4 (for each transition $s \rightarrow s'$ do) in the algorithm in Fig. 3 is such that: For each two transitions $s \rightarrow s'$, $s \rightarrow s''$, it is the case that $s \rightarrow s'$ is chosen before $s \rightarrow s''$ iff in the same line in the algorithm in Fig. 2 the transition $rep(s) \rightarrow s_1$, where $rep(s_1) = rep(s')$, is chosen before the transition $rep(s) \rightarrow s_2$, where $rep(s_2) = rep(s'')$.

Then, when applied on $T$, for both algorithms the total number of states that are removed from unexpanded in line 3 is the same, as well as the total number of transitions that are explored in line 4.

Proof. For brevity, we denote the algorithms in Fig. 2 and 3 with A3 and A4, respectively. In order to prove the property we show that each execution step of A3 can be mimicked by an execution step of A4. To this end we prove that at each point the following invariants hold:

1. the set of reached states is the same in both executions,
2. there is a bijection between the sets unexpanded such that each state $s$ from unexpanded of A4 is mapped into the state $rep(s)$ in unexpanded of A3.

Obviously the invariants are preserved after the initialization of reached and unexpanded in line 1. Constraint (1) and invariant (2) imply that, if a state $s$ is chosen in line 3 of A3, then a state $s'$ is chosen in the same line of A4 such that $rep(s') = s$. Similarly, because of constraint (2), if a transition $s \rightarrow s_1$ is chosen in line 4 of A3, then the transition $s' \rightarrow s'_1$ is chosen in the same line of A4 such that $s = rep(s')$ and $s_1 = rep(s'_1)$. (As any symmetry is bijection, the same number of transitions are chosen in both executions in line 4.) Thus, from the discussion above it follows that, if an error is reported (line 5) by A3, then it is reported also by A4. Further, having in mind invariant (1) we can conclude that the boolean expression "$rep(s') \notin reached" (line 7) has the same value in both executions. This in combination with constraint (2) also implies that the invariants are preserved by the statements in line 8 that update reached and unexpanded.

If $rep$ is not canonical, then the number of explored states and transitions differs between the two algorithms and depends on the choice of $rep$. The algorithms are incomparable in this case, because it can happen that our algorithm explores fewer states and/or transitions than the algorithm in Fig. 2, or vice versa.

However an advantage of our algorithm is that it allows (regardless whether $rep$ is canonical or not) to easily regenerate an erroneous trace from the set
unexpanded, in case an error is encountered. Since Spin explores the state space in a depth first manner, the set unexpanded is in fact structured as a stack. When an error is encountered the stack contains the sequence of states that lead to the error, and its contents can directly be dumped as the erroneous trace. In the algorithm from Fig. 2, the stack would contain the representatives of the original states, and since the representatives are not necessarily related by the transition relation in the original model, the stack would not necessarily represent an existing trace in the original model.

We put an effort into efficiently implementing the 5 variants of the rep function. For example, it turns out that all the sets of pid permutations used in our reduction strategies can always be enumerated starting with an initial permutation and then composing it with transpositions (permutations that swap two elements). As a consequence, the most costly operation \( p^* \) (that applies a given pid permutation to a given state) can be optimized by using two versions of \( p^* \). In addition to the general version that is applicable to any pid permutation, we also use a restricted (and more efficient) version that is only applicable to transpositions. Our implementation is described in more details below.

In the verification experiments described in Section 5, we used Spin in two ways: with and without its partial order reduction (POR) algorithm. Allowing Spin to use its POR algorithm together with our symmetry reductions is sound due to Theorem 19 in [9] which guarantees that the class of POR algorithms to which the Spin’s POR algorithm belongs, is compatible with the generic symmetry reduction algorithm. With a straightforward modification, the theorem’s proof is valid for our algorithm as well.

It can be shown (see next chapter) that the nested depth-first search algorithm of [6], which is used in Spin for cycle detection, remains correct with the symmetry reduction. This implies that we can go beyond safety properties, or more precisely, the full class of \( \omega \)-regular correctness properties can be handled by Spin.\(^4\)

4.2 An Overview of SymmSpin

Our goal was first to experiment with various reduction strategies, to check whether they perform well enough to undertake the effort of fully extending Spin with a symmetry package (such as modifying the Promela parser). So the problem was how to extend Spin in a minimal way that would be sufficient to perform various verification experiments with some symmetric protocols. In fact, we have managed to find a way which does not require any change to the Spin tool itself, but only to the C program generated by Spin.

The C program generated by Spin is kept in several files. Two of them are of particular interest to us: \texttt{pan.c} that implements a state exploration algorithm, and \texttt{pan.h} that contains the declarations of various data structures used

\(^4\) In fact, as pointed out in [4, 10], the property which is being checked should also be symmetric. More precisely, the property should be invariant under any permutation of the scalarsets (Pids).
in \texttt{pan.c}, including a C structure called \texttt{State} that represents a state of the transition system being verified. Our current extension of Spin with symmetry reductions simply adds a particular reduction strategy to \texttt{pan.c}. This is done in two steps.

First, we locate in \texttt{pan.c} all calls to the procedures that add a newly visited state to the set of already visited states. The calls have the generic form \texttt{store(now)} where \texttt{now} is a global variable that keeps the current state. All the calls are changed to \texttt{store(rep(now))} where \texttt{rep} is the name of a C function that implements the reduction strategy. In this way, the representatives, and not the states themselves, are stored in the set of already visited states. The \texttt{pan.c} code that computes the successor states of \texttt{now} is left unchanged. As a consequence, the original states, and not their representatives, are used to generate the state space to be explored. This agrees with the exploration algorithm in Fig. 3.

Second, the C code for \texttt{rep} is generated and added to \texttt{pan.c}. This step is not straightforward since we have to scan \texttt{pan.h} for various pieces of information needed for the implementation of \texttt{rep}.

The two steps are performed by a Tcl script called \texttt{AdjustPan}. The current version of the script is about 1800 lines, not counting comments.

4.3 The \texttt{AdjustPan} Script

Conceptually, the symmetry relation is deduced from a Promela program, say \texttt{foo.pml}, that uses special types called \texttt{scalarsets} which are unordered ranges of integers (in SymmSpin, always of the form 0..n - 1, for some constant \(n < 256\)).

Actually, in order to avoid modifications to the Promela parser, there is no special declaration for scalarsets. Instead, the standard Promela type \texttt{byte} is used for this purpose. Also, the symmetry relation is not deduced from \texttt{foo.pml} itself. Instead, it is deduced from an additional file, say \texttt{foo.sym}, that must accompany the Promela program. The additional file (later called a \texttt{system description file}) must be prepared by a user, and must contain all information relevant to the usage of scalarsets in the original Promela program that specifies a symmetric concurrent system.

The precise description of the syntax of the system description file, and its meaning, is beyond the scope of this paper. In short, it resembles the declaration part of Promela, and allows to describe all the data structures and processes, appearing in the original Promela program, that depend on scalarsets. This description is then used by the \texttt{rep} function to locate all fragments of the Spin state vector that depend on a particular scalarset, and lift a permutation of the scalarset to the state.

The only important restriction in the current version of SymmSpin is that the concurrent system specified by a symmetric Promela program must be static, in the sense that all the processes must be started simultaneously, by the \texttt{init\{atomic\{...\}\}} statement.

The \texttt{AdjustPan} script is called with two parameters: the name of one of the 5 reduction strategies described in Section 3, and the name of a system.
description file. The script reads three files: `pan.h`, `pan.c` (both generated by Spin from `foo.pml`), and `foo.sym`. The information in `foo.sym` and `pan.h` is used to modify `pan.c`. The modified version of `pan.c` is stored under the name `pan-sym.c`, and is used to model check the symmetric Promela program.

In summary, SymmSpin is used in the following way:

- Write a symmetric Promela program, say `foo.pml`, and its system description file, say `foo.sym`.
- Run Spin (with the `-a` option) on `foo.pml` (this will generate `pan.h` and `pan.c` files).
- Run `AdjustPan` on `foo.sym` with a particular reduction strategy (this will generate `pan-sym.c` file).
- Compile `pan-sym.c` with the same options you would use for `pan.c`, and run the generated binaries to model check the symmetric Promela program.

### 4.4 The Implementation of Strategies

In this section we highlight some implementation details of SymmSpin, to give an idea of how the `rep` function is implemented in an efficient way. To simplify the presentation, we assume that only one scalarset is used in a Promela program.

The canonical strategies `full`, `segmented` and `pc-segmented` are implemented by a C function of the following shape:

```c
State tmp_now, min_now;

State *rep(State *orig_now) {
    /* initialize */
    memcpy(&tmp_now, orig_now, vsize);
    /* find the representative */
    memcpy(&min_now, &tmp_now, vsize);
    ...
    return &min_now;
}
```

The parameter `orig_now` is used to pass the current state `now`. In order to avoid any interference with the original code in `pan.c` that uses `now` for its own purposes (for example, to generate the successor states of the current state), we must assure that `rep` does not modify `now`. For this reason, we copy `orig_now` to the auxiliary state `tmp_now`.

The representative is found by enumerating permutations of a scalarset, and applying them to (the copy of) the current state. The lexicographically smallest result is kept in `min_now`. After each permutation, it is updated by the following statement

```c
if (memcmp(&tmp_now, &min_now, vsize) < 0)
    memcpy(&min_now, &tmp_now, vsize);
```

In strategies `sorted` and `pc-sorted`, only one permutation is considered, so the auxiliary state `min_now` is not needed. Hence, the strategies are implemented by a function of the following shape:
State tmp_now;

State *rep(State *orig_now) {
    /* initialize */
    memcpy(&tmp_now, orig_now, vsize);
    /* find the representative */
    ...
    return &tmp_now;
}

In the rest of this section, we present the rep functions for the full, sorted and segmented strategies. The pc-sorted and pc-segmented strategies are similar to the sorted and segmented strategies.

Strategy full

State tmp_now, min_now;

State *rep_full(State *orig_now) {
    memcpy(&tmp_now, orig_now, vsize);
    /* find the representative */
    memcpy(&min_now, &tmp_now, vsize);
    permute_scalar(SIZE_OF_SCALAR);
    return &min_now;
}

The representative is found by the permute_scalar procedure that takes the size of a scalarset, say size, and generates all permutations of numbers from 0 to size − 1, excluding the identity permutation. Each permutation is then applied to the current state. Applying a permutation may be quite expensive. It can be implemented more efficiently if a permutation is a transposition (i.e., a swap of two numbers). For this reason, the permutations are generated incrementally, by composing successive transpositions (starting from the identity permutation). A tricky algorithm (borrowed from [21], and presented in Fig. 4) is used for this purpose.

It happens that the transpositions generated by the algorithm always swap two successive elements p and p + 1, but we do not rely on this feature. Whenever a new transposition is computed, permute_scalar calls apply_swap. The apply_swap procedure has the following header:

void apply_swap(State *state, int v1, int v2)

It lifts the transposition of scalar values v1 and v2 to a given state. The lifting is performed in situ, by modifying the given state. The body of apply_swap is generated by AdjustPan, using the information given in a system description file. The generated C code is straightforward. It consists of a sequence of guarded assignments that swap v1 with v2, for each variable that depends on a scalarset.

For example, if x is a global variable of a scalarset type then the following code fragment is generated:
void permute_scalar(int size) {
    int i, p, offset,
        pos[MAX_SCALAR_SIZE], dir[MAX_SCALAR_SIZE];

    for (i = 0; i < size; i++) {
        pos[i] = 1; dir[i] = 1;
    }
    pos[size-1] = 0;

    i = 0;
    while (i < size-1) {
        for (i = offset = 0; pos[i] == size-i; i++) {
            pos[i] = 1; dir[i] = !dir[i];
            if (dir[i]) offset++;
        }
        if (i < size-1) {
            p = offset-1 +
                (dir[i] ? pos[i] : size-i-pos[i]);
            pos[i]++;

            /* apply transposition p <-> p+1 */
            apply_swap(&tmp_now, p, p+1);
            if (memcmp(&tmp_now, &min_now, vsize) < 0)
                memcpy(&min_now, &tmp_now, vsize);
        }
    }
}

Fig. 4. Generating permutations by transpositions
if (state->x == v1) state->x = v2; else
if (state->x == v2) state->x = v1;

For arrays, the code is more complex. For example, if \( x \) is a global array of a scalarset type, and indexed by the same scalarset, then the following code fragment is generated:

```c
/* swap values */
for (i = 0; i < SCALAR_SIZE; i++) {
    if (state->x[i] == v1) state->x[i] = v2; else
    if (state->x[i] == v2) state->x[i] = v1;
}

/* swap indices */
{ uchar tmp;
    tmp = x[v1]; x[v1] = x[v2]; x[v2] = tmp; }
```

In addition, for every family of processes indexed by a scalarset, say \( \text{proctype } P(scalar \ i) \), \text{apply\_swap} swaps the two chunks of memory (in a state vector) that correspond to \( P(v_1) \) and \( P(v_2) \).

**Strategy sorted**

```c
State *rep_sorted(State *orig_now) {
    int perm[MAX_SCALAR_SIZE];
    memcpy(&tmp_now, orig_now, vsize);
    /* sort the main array and compute
     * the sorting permutation */
    ...
    /* find the representative */
    apply_perm(perm, &tmp_now);
    return &tmp_now;
}
```

The main array is sorted using a straightforward algorithm that successively finds minimal elements. Its quadratic complexity is acceptable since the size of a scalarset is usually small. On the other hand, it allows to compute the sorting permutation with minimal cost, since each element is swapped only once.

The \text{apply\_perm} procedure has the following header:

```c
void apply_perm(int *perm, State *state)
```

It lifts the given permutation of scalarset values to \text{state}. As in \text{apply\_swap}, the lifting is performed \textit{in situ}, and it consists of a sequence of guarded assignments, for each variable that depends on a scalarset. For example, if \( x \) is a global variable of a scalarset type then the following code fragment is generated:

```c
if (state->x < SCALAR_SIZE)
    state->x = perm[state->x];
```
The guard is needed to solve a subtle problem with the initialization of scalarset variables. Since all variables used in a standard Promela program are automatically initialized with 0, it is common to use 0 to represent an undefined value. Unfortunately, this convention cannot be used for scalarsets. The reason is that in our implementation, a scalarset of size \( n \) is a range of integers from 0 to \( n - 1 \), so 0 must be treated as other well-defined values (otherwise, the symmetry would be broken). Thus, a value outside of the range must be used for an undefined value. By convention, we use \( n \) for this purpose, and the guard guarantees that the undefined value is treated in a symmetric way (i.e., it is never permuted, as required in [19]). In fact, any value not less than \( n \) can be used to represent the undefined value.

For arrays, the code is more complex. For example, if \( x \) is a global array of a scalarset type, and indexed by the same scalarset, then the following code fragment is generated:

```c
/* permute values */
for (i = 0; i < SCALAR_SIZE; i++) {
    if (state->x[i] < SCALAR_SIZE)
        state->x[i] = perm[state->x[i]];
}
/* permute indices */
{ uchar buf[SCALAR_SIZE];
    memcpy(buf, state->x, sizeof(state->x));
    for (i = 0; i < SCALAR_SIZE; i++)
        state->x[perm[i]] = buf[i];
}
```

Notice that when permuting indices we have to use a buffer.

**Strategy segmented**

```c
State *rep_segmented(State *orig_now) {
    int perm[MAX_SCALAR_SIZE];
    memcpy(&tmp_now, orig_now, vsize);
    /* sort the main array and compute
     the sorting permutation */
    ...  
    /* locate blocks */
    ...  
    /* find the representative */
    apply_perm(perm, &tmp_now);
    memcpy(&min_now, &tmp_now, vsize);
    if (num_of_blocks > 0)
        permute_blocks(0, block_start[0],
                       block_size[0]);
    return &min_now;
}
```

First, the main array is sorted as in the **sorted** strategy. Second, the segments of equal values are located, in the sorted main array, by a straightforward linear
The information about the segments (called blocks henceforth) is stored in the following global data structures:

```c
int num_of_blocks;
int block_start[MAX_SCALAR_SIZE],
    block_size[MAX_SCALAR_SIZE];
```

Finally, the canonical representative is found by procedure `permute_blocks` that generates all permutations of indices in successive blocks, excluding the identity permutation. Its code is given in Fig. 5. Each permutation is then applied to the current state, and the lexicographically smallest result is chosen.

```c
void permute_blocks(int block, int start,
                     int size) {
    int i, p, offset,
        pos[MAX_SCALAR_SIZE], dir[MAX_SCALAR_SIZE];

    /* go to the last block */
    if (++block < num_of_blocks)
        permute_blocks(block, block_start[block],
                        block_size[block]);
    block--;

    /* the same as permute_scalar, but apply
     * transposition is changed to */
    ...
    swap_in_block(block, p, p+1);
    ...
}
```

```c
void swap_in_block(int block, int p1, int p2) {
    /* apply transposition p1 <-> p2 */
    apply_swap(&tmp_now, p1, p2);
    if (memcmp(&tmp_now, &min_now, vsize) < 0)
        memcpy(&min_now, &tmp_now, vsize);
    /* permute the next block */
    if (++block < num_of_blocks)
        permute_blocks(block, block_start[block],
                        block_size[block]);
}
```

**Fig. 5.** Permuting blocks

The procedure uses double recursion, to assure that the number of calls to `apply_swap` and comparisons between `tmp_now` and `min_now` is minimal. It is a generalization of `permute_scalar` used in `rep_full`. Conceptually, the indices of separate blocks, when considered relative to the start of a respective block,
can be perceived as separate scalarsets. Inside one block, all transpositions of the block’s indices are generated as in `permute_scalar`. The double recursion is used to properly compose the permutations of separate blocks, by chaining the invocations of `permute_blocks`.

5 Experimental Results

We tried our prototype implementation on several examples. The obtained reductions were often very close to the theoretical limits, thus, we were able to obtain reductions of several orders of magnitude in the number of states. Also, due to the significantly smaller number of states that had to be explored, in all the cases the verification with symmetry reduction was faster than the one without it. The experiments showed that there is no favorite among the reduction strategies regarding the space/time ratio. This suggests that it makes sense to have all strategies (maybe except full) as separate options of the extended model-checker.

In the verification experiments we used Spin and SymmSpin both with and without the partial order reduction (POR) option. In most of the cases there was a synergy between the symmetry and the partial order reductions. The two reduction techniques are orthogonal because they exploit different features of the concurrent systems, therefore, their cumulative effect can be used to obtain more efficient verification.

In the sequel, we present the results for three of the examples. All experiments were performed on a Sun Ultra-Enterprise machine, with three 248 MHz Ultra SPARC-II processors and 2304MB of main memory, running the SunOS 5.5.1 operating system. Verification times (in the rows labeled with “t”) are given in seconds (s.x), minutes (m:s), or hours (h:m:s); the number of states (in the rows labeled with “s”) is given directly or in millions (say, 9.1M); o.m. stands for out of memory, and o.t. denotes out of time (more than 10 hours); +POR and -POR mean with and without POR, respectively.

5.1 Peterson’s Mutual Exclusion algorithm [22].

For this well known example\(^5\) we verified the mutual exclusion property. The results for different numbers \(N\) of processes are shown in Table 1.

The gain due to the symmetry reduction is obvious. The obtained reductions, ranging from 49% (for \(N = 2\)) to 99% and more (for \(N \geq 5\)) are close to the theoretical maxima, which can be explained with the high degree of symmetry in the protocol. The verification times are also better, even with the straightforward full strategy, due to the smaller number of states that are generated during the search. We used both symmetry and partial order reduction (which is default

---

\(^5\) In our implementation the global predicate that guards the entry in the critical section is checked atomically. As this guard ranges over all process indices, the atomicity was necessary due to the restrictions on statements that can be used such that the state space symmetry is preserved.
Table 1. Results for Peterson's mutual exclusion algorithm.

<table>
<thead>
<tr>
<th>N</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>POR</td>
<td>POR</td>
<td>POR</td>
<td>POR</td>
<td>POR</td>
<td>POR</td>
</tr>
<tr>
<td>no sym.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>154</td>
<td>263</td>
<td>4992</td>
<td>11318</td>
<td>202673</td>
<td>542921</td>
</tr>
<tr>
<td>t</td>
<td>6.5</td>
<td>5.6</td>
<td>6.8</td>
<td>6.3</td>
<td>25.0</td>
<td>1:09</td>
</tr>
<tr>
<td></td>
<td>19:04</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>full</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>89</td>
<td>134</td>
<td>980</td>
<td>1976</td>
<td>9634</td>
<td>24383</td>
</tr>
<tr>
<td>t</td>
<td>6.5</td>
<td>5.6</td>
<td>6.7</td>
<td>5.9</td>
<td>10.4</td>
<td>19.3</td>
</tr>
<tr>
<td></td>
<td>2:35</td>
<td>12:02</td>
<td>2:04:03</td>
<td>o.t.</td>
<td>o.t.</td>
<td>—</td>
</tr>
<tr>
<td>seg.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>113</td>
<td>160</td>
<td>1877</td>
<td>3967</td>
<td>35644</td>
<td>88489</td>
</tr>
<tr>
<td>t</td>
<td>6.5</td>
<td>5.6</td>
<td>6.6</td>
<td>5.8</td>
<td>8.5</td>
<td>12.4</td>
</tr>
<tr>
<td></td>
<td>38.0</td>
<td>2:31</td>
<td>7:36</td>
<td>44:40</td>
<td>1:56:00</td>
<td>—</td>
</tr>
<tr>
<td>pc-seg.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>92</td>
<td>137</td>
<td>1149</td>
<td>2396</td>
<td>20339</td>
<td>46804</td>
</tr>
<tr>
<td>t</td>
<td>6.5</td>
<td>5.6</td>
<td>6.6</td>
<td>5.8</td>
<td>10.2</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>1:36</td>
<td>4:53</td>
<td>54:53</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

The number of states is obtained with segmented and pc-segmented.

in standard Spin), separately and in combination. Standard Spin with partial order reduction could not handle more than 5 processes. However, the segmented versions of the symmetry heuristics alone were sufficient for $N = 6$. For $N = 7$ we had to use a combination of both reduction techniques in order to stay inside the available memory.

The sorted strategies are comparable with their segmented counterparts only for small values of $N$. For greater values they deteriorate and even the possible gain in time over the segmented versions disappears because of the greater number of states that have to be explored.

One can also expect that as $N$ increases pc-segmented and pc-sorted will loose the advantage they have over segmented and sorted, for smaller values of $N$. The reason is that the number of different elements in the main array of segmented and sorted increases as $N$ increases, while the number of values of the pc counter stays the same. (We use as a main array the array of flags. Whenever process $i$ enters the competition for the critical section, it sets flag $i$ to a value between 1 and $N - 1$. The default value is 0. Notice that the values of the flag array are not of the scalarset type $Pid$, although their range is the same as the range of $Pid$.) Intuitively, the greater versatility of the values in the main array, the fewer permutations have to be generated on average, in order to canonicalize the state. This tendency is already visible for $N = 6$ (for which sorted is winning over pc-sorted) as well as for $N = 7$ (for which segmented is better than pc-segmented).

5.2 Data Base Manager [24].

The system that we consider consists of $N \geq 2$ data base managers, which modify a data base and exchange messages to ensure the consistency of the data base
contents. Our model deals with the procedural part of the protocol, i.e., with the message exchange, by abstracting from the actual modification of the data base. Initially all managers are in inactive state until one of them modifies the data base. This manager in one atomic step reserves the data base for itself and sends a message to every other manager. After that it waits for acknowledgments from the other managers. All other managers concurrently perform a two step sequence: reception of the message, and sending of an acknowledgment. When all acknowledgments are available, the manager who initially modified the data base and started the whole procedure, reads them. At the same moment it also releases the data base so that it can be modified by the other managers, after which it returns to inactive state. We checked the model for absence of deadlock.

**Table 2.** Results for the Data Base Manager example.

<table>
<thead>
<tr>
<th>N</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>100</td>
<td>5112</td>
<td>130</td>
<td>17506</td>
<td>164</td>
<td>59060</td>
</tr>
<tr>
<td>t</td>
<td>0.0</td>
<td>0.5</td>
<td>1.9</td>
<td>8.2</td>
<td>0.0</td>
<td>32.1</td>
</tr>
<tr>
<td>no sym.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>16</td>
<td>31</td>
<td>18</td>
<td>39</td>
<td>48</td>
<td>22</td>
</tr>
<tr>
<td>t</td>
<td>0.5</td>
<td>2.0</td>
<td>4.6</td>
<td>24.5</td>
<td>48.0</td>
<td>6:16</td>
</tr>
<tr>
<td>full</td>
<td>seg.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>0.3</td>
<td>0.4</td>
<td>2.8</td>
<td>3.8</td>
<td>28.3</td>
<td>39.1</td>
</tr>
<tr>
<td>pc-seg.</td>
<td>t</td>
<td>0.1</td>
<td>0.1</td>
<td>0.8</td>
<td>1.0</td>
<td>7.3</td>
</tr>
<tr>
<td>sorted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>27</td>
<td>250</td>
<td>31</td>
<td>505</td>
<td>35</td>
<td>1016</td>
</tr>
<tr>
<td>t</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>pc-sort.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>16</td>
<td>58</td>
<td>18</td>
<td>91</td>
<td>20</td>
<td>155</td>
</tr>
<tr>
<td>t</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Our experiments are in accord with the theoretically predicted results from [24]: both symmetry and POR applied separately reduce the exponential growth of the state space to quadratic, and in combination they give linear growth. Unlike in the previous example, this time pc-segmented strategy is a clear winner over segmented. The explanation is again in the diversity of the elements occurring in the sorted array – the only possible values of the main array for segmented are 0 and 1, while the pc counter has 7 different values. (The main array consists of boolean flags which are set to 1 when a manager reserves the data base and sends a message to all the other processes.) It is interesting that pc-sorted in combination with POR is by far the most successful strategy. It achieves the same reduction as the canonical strategies, but within much shorter time. It remains to be seen whether this is just a peculiarity of this model, or it occurs for a wider class of examples.
5.3 The Initialization Protocol [14].

Assume that we are given a system of \(n\) processes each consisting of an initial part and a main part. The data structures can have arbitrary initial values. The role of the Initialization Protocol from [14] is to synchronize the processes such that none of them can enter its main part before all the others have finished their initial parts.\(^6\) Process synchronization is done via shared boolean variables. Each process has so called co-component process whose task is to manage the synchronization of the boolean variables. We refer the reader to [14] for more details.

The protocol is an example of a system with one scalar variable and two process families indexed by this scalar – the family of processes that are synchronized and the family of their co-components. In such a case one can expect the same maximal amount of reduction of \(n!\) (where \(n\) is the family size) as with one family.

| Table 3. Results for the Initialization Protocol example. |
|---|---|---|---|---|---|
| \(n\) | 2 | 3 | 4 | 5 |
| --- | --- | --- | --- | --- | --- |
| no sym. | 2578 | 131484 | 7.0M | o.m. | o.m. |
| | 6.4 | 19.3 | 18:50 | --- | --- |
| full | 1520 | 26389 | 386457 | 5.3M |
| | 6.2 | 6.0 | 4:53 | 5:38:05 | --- |
| seg. | 6.3 | 11.1 | 3:19 | 2:28:20 | --- |
| pc-seg. | 6.3 | 10.1 | 1.54 | 47:03 | --- |
| sorted | 2344 | 98602 | 4.2M | o.m. | o.m. |
| | 6.3 | 17.9 | 12:13 | --- | --- |
| pc-sort. | 1607 | 40395 | 987830 | o.m. | o.m. |
| | 6.3 | 11.8 | 3:55 | --- | --- |

We verified the correctness of the algorithm for several values of \(n\) by placing before the main part of each process an assertion which checks that all the other processes have terminated their initial parts.

Because of the intensive use of global variables, partial order reduction had almost no effect in this example. Thus, the state space reduction was mostly due to the symmetry. The canonical heuristic segmented and pc-segmented prevailed over the non-canonical ones regarding both space and time. Compared to full, they were significantly faster for larger values of \(n\). Better performance of pc-segmented and pc-sorted over segmented and sorted respectively, can be explained.

\(^6\) Note that this would be trivial if we could assume that the variables’ initial values were known.
by the fact that the main array that was used in the latter strategies is of boolean
type. This means that the elements of the main array can have only two values,
in contrast with the program counter which ranges over at least 15 different
values. Intuitively, the greater versatility of values in the pc array means that
fewer permutations have to be generated on average in order to canonicalize the
state.

5.4 Base Station.

This example is a simplified version of MASCARA – a telecommunication proto-
col developed by the WAND (Wireless ATM Network Demonstrator) consortium
[7]. The protocol is an extension of the ATM (Asynchronous Transfer Mode) net-
working protocol to wireless networks. We present the results for two models of
the protocol.

Our first model represents a wireless network connecting \( N \geq 2 \) mobile sta-
tions (MS) that may communicate with each other, using a limited number
\( (M \geq 1) \) of radio channels provided by one base station BS. More specifically,
when MS \( A \) wants to send a message to MS \( B \) it must request a channel from
BS. Provided there are channels available, \( A \) is granted one, call it \( c \). If \( B \) wants
to receive messages, it queries BS. As there is a pending communication for \( B 
\) through \( c \), BS assigns \( c \) to \( B \). After the communication has taken place, both
\( A \) and \( B \) return the channel to BS. The results given in Table 4 are for checking
for unreachable code, with \( M = 2 \).

<table>
<thead>
<tr>
<th>( N )</th>
<th>( 2 )</th>
<th>( 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-POR</td>
<td>-POR</td>
</tr>
<tr>
<td>no sym.</td>
<td>s 15613</td>
<td>15621</td>
</tr>
<tr>
<td>full</td>
<td>s 7808</td>
<td>7812</td>
</tr>
<tr>
<td>seg.</td>
<td>t 7.5</td>
<td>6.3</td>
</tr>
<tr>
<td>pc-seg.</td>
<td>t 5.9</td>
<td>6.4</td>
</tr>
<tr>
<td>sorted</td>
<td>s 7856</td>
<td>7860</td>
</tr>
<tr>
<td>pc-sort.</td>
<td>s 8282</td>
<td>8286</td>
</tr>
</tbody>
</table>

For \( N = 2 \) the symmetry reduction approaches the theoretical limit of 50%.
For \( N = 3 \) standard Spin ran out of memory, while with symmetry reduction it
was possible to verify the model. In this example POR did not play a significant
role. Also, there was no clear preference between segmented, pc-segmented and sorted.

In the second model, instead of having $N$ stations which can both receive and send messages, we consider a network connecting $n_s \geq 1$ sending mobile stations and $n_r \geq 1$ receiving mobile stations. Another important difference is that the communication is modeled with Promela channels, instead of global variables. For the rest the description of the protocol given for the first model applies also to the second one.

### Table 5. Results for the Base Station example (second model).

<table>
<thead>
<tr>
<th></th>
<th>$n_s = 2$</th>
<th>$n_s = 3$</th>
<th>$n_s = 2$</th>
<th>$n_r = 3$</th>
<th>$n_s = 2$</th>
<th>$n_r = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>no sym</td>
<td>s</td>
<td>4.6M</td>
<td>9.0M</td>
<td>o.m.</td>
<td>o.m.</td>
<td>o.m.</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>5:02</td>
<td>13:48</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>full</td>
<td>s</td>
<td>1.2M</td>
<td>2.3M</td>
<td>9.0M</td>
<td>9.4M</td>
<td>o.m.</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>1:41</td>
<td>5:03</td>
<td>28:57</td>
<td>30:32</td>
<td>—</td>
</tr>
<tr>
<td>pc-seg.</td>
<td>s</td>
<td>1.7M</td>
<td>3.5M</td>
<td>o.m.</td>
<td>o.m.</td>
<td>o.m.</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>1:58</td>
<td>6:03</td>
<td>39:02</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>pc-sort.</td>
<td>s</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Thus, there are two families of symmetric processes in the second model: The family of sending stations and the family of receiving stations. Unlike in the Initialization Protocol, the two families are independent, and, consequently, indexed with two different scalar sets.

On this example we could apply only pc-segmented and pc-sorted because there was no main array that could be used for sorted and segmented. One can see that for the canonical strategies the reduction indeed approaches the theoretical limit $n_s! \cdot n_r!$. (In general, for a system with $k$ independent process families indexed by $k$ scalar sets, the reduction due to symmetry is limited by $n_1! \cdot n_2! \cdot \ldots \cdot n_k!$, where $n_i, 1 \leq i \leq k$, is the number of processes in the $i$-th family.)

With the usage of Promela channels instead of global variables, the contribution of the partial order reduction became significant (Table 5), which was not at all the case in the first model. One can also notice the orthogonality of the two heuristics – namely, the factor of reduction due to symmetry was almost the same with or without POR.
6 Conclusions and Future Work

We have presented four variants of a new heuristic for coping with the orbit problem in symmetry reduction. Their strength is based on the observation that the lexicographical ordering on state vectors may be tuned to optimize the splitting heuristic of [18]. Of the resulting strategies, the segmented/pc-segmented versions indeed produce canonical representatives of orbits, while sorted/pc-sorted are only normalizing. The sorted and segmented strategies presuppose the presence of a main array in the program. Their pc variants exploit the array of program counters in programs containing a family of similar processes.

Based on these strategies, we developed SymmSpin, an extension of Spin with a symmetry-reduction package. We compared the effects of the strategies, as well as the reference full strategy, on several examples. The experimental results show that there is no uniform winner: It depends on the program which of the strategies performs best. In some cases, this was in fact predicted by the form of the program. Thus, it make sense to have all strategies (maybe except full) as separate options of the extended model-checker.

The resulting package has been described at a detailed level. For maximal modularity, the implementation is in the form of a Tcl script that operates on the verification engine produced by Spin. We point the interested reader to the web site accompanying this paper, [1], for the full source code.

The results obtained with this experimental implementation are sufficiently encouraging in order to integrate the symmetry strategies more completely into the Spin tool. One concern here is to ensure the compatibility of the symmetry reduction with Spin’s other features. So far, we have focussed on the partial order reduction. In the next chapter we show that the correctness of Spin’s cycle detection algorithm, which lies at the heart of its capability to handle the full class of $\omega$-regular correctness properties, is preserved. Furthermore, the symmetry is compatible with the recent extension to timed systems, DTSpin ([2]). It should be added that the present implementation is able to deal with multiple scalar sets and multiple process families. Also, almost all Promela features are handled, including queues. For the latter there is still the restriction though that the queue elements have to be simple, non-structured types.

Also, the restrictions on the scalarset variables are as yet not automatically checked. The most natural solution to this problem would be to extend the Promela syntax to allow specifications of scalar sets directly instead of via an accompanying file. This will facilitate the automatic verification of the syntactic conditions along the lines of [19], that are needed to ensure the soundness of the analysis. With the present implementation this is still a task for the user. A more ambitious attempt would be the automatic detection of scalar sets directly from the Promela sources.
Next, we plan to extend the symmetry algorithm itself with other system topologies, like rings. An interesting question is whether the choice of an appropriate strategy can be automated, based on syntactic clues.

References


7 The group of shift permutations can be handled by a simplification of our approach, where instead of sorting the main array, it is cyclically shifted until it starts with a minimal element.
A Heuristic for Symmetry Reductions with Scalarsets


7. Nested Depth First Search Algorithms for Symmetry Reduction in Model Checking
Nested Depth First Search Algorithms for Symmetry Reduction in Model Checking

Dragan Bošnački

Dept. of Computing Science, Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: dragan@win.tue.nl

Abstract. We propose an algorithm for model checking under weak fairness that exploits symmetry for state space reduction. As intermediate results we also discuss two other algorithms which deal separately with weak fairness and symmetry reduction. The algorithms presented in this chapter are based on the Nested Depth First Search (NDFS) algorithm by Courcoubetis, Vardi, Wolper and Yannakakis. We argue that the worst case time complexity of our algorithm for model checking under weak fairness with symmetry is of the same order of magnitude as the time complexity of the algorithms known in the literature. Moreover, as these algorithms require finding all strongly connected components (SCC) in the state space graph, our algorithm preserves the advantages of the NDFS over the SCC approach.

1 Introduction

Model checking [6, 22] is a widespread technique for the automated verification of concurrent systems. The technique lends itself to implementation and as such it is successfully used in the construction of verification tools [23]. However, model-checking tools are often limited by memory requirements because of the problem of state space explosion. One of the important techniques to alleviate this problem is symmetry reduction.

Symmetry is present in many systems, like mutual exclusion algorithms, cache coherence protocols, bus communication protocols, etc. In order to grasp the idea behind symmetry reduction, consider a typical mutual exclusion protocol. The (in)possibility for processes to enter their critical sections simultaneously will stay the same if the process identities are permuted. As a consequence, when during state-space exploration a state is visited that is the same, up to a permutation of pids, as some state that has already been visited, the search can be pruned. More formally, the symmetry of the system is represented by a given group $G$ of permutations that act on the global states of the system. It turns out that two states $s$ and $s'$ are behaviorly equivalent if there exists a permutation $\pi \in G$ such that $s'$ can be obtained by applying $\pi$ on $s$. Thus, the system state space $T$ is partitioned into equivalence classes. We define a selection function $h$ which selects a unique representative from each equivalence class. Next, the quotient state space $h(T)$ is constructed that contains only these
representatives and the property is checked using $h(T)$ instead of $T$. As $h(T)$ is in general much smaller than $T$, the gain in memory and time needed for the verification algorithms can be significant.

The issue of fairness is an inherent and important one in the study of concurrency and nondeterminism, in particular in the area of the verification of concurrent systems. Since fairness is used as generic notion there is a broad taxonomy of fairness concepts. In this chapter we confine our attention to the notion of weak fairness on the level of processes. This means that we require that for every execution sequence of the concurrent program which is a composition of several processes, if some process becomes continuously enabled at some point of time (i.e. can always execute some of its statements), then at least one statement from that process will eventually be executed. This kind of fairness is most often associated with mutual exclusion algorithms, busy waiting, simple queue-implementations of scheduling, and resource allocation. Weak fairness will guarantee the correctness of statements like eventually entering the critical region for every process which is continuously trying to do this (in the mutual exclusions) or eventually leaving the waiting queue for each process that has entered it (in the scheduling) [13].

Thus, combining the algorithms for model-checking under weak fairness with reduction techniques, and in particular symmetry, is a prerequisite for the verification of many interesting properties in practice. However, when coupling the two concepts special care should be taken, because of the possible incompatibilities between the particular algorithms.

The main contribution of the chapter is an algorithm for model checking under weak fairness (in the sense described above) that exploits symmetry reduction. As intermediate results we also discuss two other algorithms which deal separately with weak fairness and symmetry reduction.

We assume that the properties that are checked are specified as Büchi automata [21]. As a consequence the problem of checking whether some given property holds for the system under consideration can be reduced to the problem of finding acceptance cycles (i.e., cycles that contain acceptance states) in the graph representing the product of the system model state space with the property automaton (c.f. [7, 5]).

All the algorithms that are presented in the chapter are based on the so-called nested depth first search (NDFS) algorithm by Courcoubetis, Vardi, Wolper and Yannakakis [7] for detecting acceptance cycles in the state space. In the NDFS algorithm from each acceptance state which is visited during the standard DFS exploration of the state space a check procedure is called in order to detect a possible acceptance cycle through the state. This nested cycle check is itself a DFS, which explains the name “nested DFS”.

In order to capture weak fairness, one has to modify the standard NDFS algorithm. This is because the latter only guarantees that it will find some acceptance cycle, if there exists one, but not all of them. Thus, one cannot use the straightforward idea to just ignore the detected cycles which are not fair until one finds a fair one, or there are no more cycles. In the chapter we present an
algorithm for model checking under weak fairness which is a minor modification of the weak fairness algorithm by Gerard Holzmann implemented in the model checker Spin. The idea of the upgrade for weak fairness is to do the acceptance cycle search in an extended state space in which it is guaranteed that each cycle that is detected is a fair one. In fact, we need a more general result that claims that there exists a fair acceptance cycle in the original state space if and only if there exists an acceptance cycle in the extended state space. We show the correctness of our modified fairness algorithm by arguing that it is equivalent to the standard NDFS algorithm applied to the extended state space.

Our second stepping stone towards the main algorithm is presented in a more general framework of state space reductions that preserve bisimulation, introduced by Emerson, Jha and Peled [12]. The symmetry based reductions are only a special case of such reductions. The algorithm that we consider is a straightforward generalization of the symmetry reduction algorithms from [19, 10, 12, 3]. After presenting an NDFS version of this generalization, we prove its correctness.

Finally, we present the main algorithm that reconciles the issues of fairness and symmetry. We begin by discussing why a straightforward combination of the NDFS versions of the fairness and bisimulation preserving reduction algorithms fails. After that we briefly introduce symmetry reduction as a special case of a bisimulation preserving reduction. The algorithm for fairness and symmetry is based on the theory developed by Emerson and Sistla [11]. In this chapter we do the necessary adjustments of the theory so that we can fit it into the NDFS concept. The main idea (borrowed from [11]) is to work with a reduced state space \( h_G(T) \) which, unlike for the ordinary symmetry reduction algorithm, has transitions annotated with permutations. In this way the process identities from the original state space \( T \), which are scrambled during the reduction, can be restored and the corresponding fair acceptance cycles detected. In order to facilitate the implementation of the cycle detection we further unfold the annotated reduced state space into a threaded state space \( h^*_G(T) \). Besides using the NDFS concept, the main novelty of our approach is to reduce the search for a fair acceptance cycle in the annotated state space \( h_G(T) \) to a search of \( N \) acceptance cycles in the threaded state space \( h^*_G(T) \), where \( N \) is the number of processes in the system. There is one to one correspondence between these acceptance cycles and the processes. Moreover, each of the cycles is weakly fair with regard to its corresponding process, meaning that the process either executes a statement along the cycle or it is disabled in some state of the cycle. The main benefit of such an approach is that one can avoid that the annotating permutations are kept as part of the representation of the state or on the DFS stack. The former is necessary in order to keep the advantages of the NDFS algorithm.

The unfolding into a threaded structure augments the size of the annotated state space roughly by a factor of \( N \). The search for fair cycles contributes an additional factor of \( N \), which gives in total \( N^2 \) times bigger structure than the reduced annotated state space. However, the gain in memory is substantial because the annotated state space is often a factor \( N! \) smaller than the original
one. In fact, using efficient storage techniques [16] one can show that in practice the factor $N^2$ can be avoided, i.e., often the threaded structure also can be stored in virtually the same memory size as the annotated state space.

In the literature we could find two algorithms for exploiting symmetry under weak fairness. The first one is from the already mentioned paper of Emerson and Sistla [11], while the second one is by Gyuris and Sistla [14] and it is an improvement of the first algorithm. We show that the time complexity of our algorithm is the same as for the above mentioned algorithms. However, as the algorithms in [11, 14] require finding all strongly connected components (SCC) in the state space graph, our algorithm capitalizes on the advantages that the NDFS approach has over the SCC based algorithms. Probably the most important one among these advantages is that, unlike the two existing algorithms, our algorithm is compatible with the memory efficient approximative verification techniques like bit-state hashing [15] and hash-compact [24]. In practice, when the property which is being verified does not hold, the NDFS based algorithms are faster and require less memory to find an error. Also, with the NDFS based algorithms it is much easier to reconstruct the counter example execution which witnesses the error, by simply dumping the contents of the stack. Our algorithm is also easier to implement because it does not require complex structures to keep the annotating permutations.\footnote{For the algorithm from [14] the authors claim that it is not necessary to keep the permutations as a part of the state. However, it is hard to see how this can be achieved without a significant time overhead.}

\textit{Chapter layout.} In the next section we give the basic notions used throughout the chapter, as well as the standard NDFS algorithm. In Section 3 we describe the weak fairness algorithm and prove its correctness using the notion of weakly fair extension of labeled transition system. Section 4 introduces bisimulation preserving reduction. In this section we give the NDFS algorithm for model checking by exploiting these kind of reductions, without fairness. In the next section we first recall the basics of symmetry reduction and the theory of [11]. After presenting the extensions of the theory we give the main algorithm that combines symmetry reduction with weak fairness, and show its correctness. Finally we discuss its space and time complexity. The last section summarizes the chapter and provides some guidelines for future work.

2 Preliminaries

2.1 Model-checking Problem, Labeled Transition systems and Bisimulations

In the sequel we adopt the automata-theoretic approach to model checking. In particular, we assume that the properties are given as Büchi automata [21]. This allows us to reduce the model checking problem to the problem of finding a cycle
in the finite graph representing the combination between the state space of the model and the property.

There are several ways to get this graph which depend on the languages in which the model and the property are specified, as well as on the formalisms used to represent their semantics. One typical way is the following:

Assume that we are given the model \( M \) and the property \( f \), represented in some modeling language and temporal logic, respectively. We want to check if all execution sequences (computations) of \( M \) satisfy \( f \). From the description of \( M \), usually given as a parallel composition of processes \( P_i \), we can obtain the global behavior of \( M \) semantically represented by a graph \( T \) whose nodes are the global states of \( M \) and the edges are the transitions between them. The negation of \( f \), \( \neg f \), is translated into the Büchi automaton \( A_{\neg f} \), which can be also regarded as a state space graph with a special set of designated states. In order to check the property we need the state space graph \( G \) which is obtained as a product of \( T \) and \( A_{\neg f} \). The intuition is that \( A_{\neg f} \) monitors \( T \) by guessing nondeterministically the infinite execution sequences in \( T \). \( A_{\neg f} \) induces designated states also in \( G \). In order for property \( f \) to hold for \( M \), the set of infinite execution sequences in \( G \) that pass through infinitely many such designated states should be empty. Otherwise, there is an execution which is in accord with the negation of \( f \), and therefore it is a counter example that the property does not hold. As \( G \) is finite, finding an infinite execution which is a counter example boils down to detecting cycles containing designated states.

The state space graph \( T \) usually grows exponentially with the size of the description of \( M \), which causes also an exponential growth of \( G \). Building the whole \( T \) can be avoided using on-the-fly techniques. In this approach we build only \( A_{\neg f} \) before starting the construction of \( G \). \( A_{\neg f} \) is used to reduce the size of \( G \) by generating only the part of \( T \) which is needed for the product, i.e., which is in accord with \( A_{\neg f} \). The property is checked simultaneously while building \( G \). This increases the efficiency of the model-checking procedure because often in practice we can come up with a counterexample in an early phase of the state space graph exploration, which means that the complete graph \( G \) does not have to be stored in the memory.

We refer the interested reader, for instance, to [7], for a formal description of the scheme given above. In the sequel we work directly with the final state space graph \( G \) representing the product of the semantics of the model and the (negation of the) property, abstracting from the way it is obtained. From the model description or the property, we keep only the information which might be needed in the verification (e.g., process IDs). For a more detailed description how the composition can be done on-the-fly see for example [5, 20, 17].

For our purposes, we represent the final state space graph as a *labeled transition system* formally defined as follows:
Definition 1. Let Prop be a set of atomic propositions. A labeled transition system (LTS) is a 6-tuple \( T = (S, R, L, A, \hat{s}, F) \), where

- \( S \) is a finite set of states,
- \( R \subseteq S \times A \times S \) is a transition relation (we write \( s \xrightarrow{a} s' \) for \( (s, a, s') \in R \)),
- \( L : S \to 2^{\text{Prop}} \) is a labeling function which associates with each state a set of atomic propositions that are true in the state,
- \( A \) is a finite set of actions,
- \( \hat{s} \) is the initial state,
- \( F \subseteq S \) is the set of acceptance states.

Unless stated differently, we fix \( T \) to be \( (S, R, L, A, \hat{s}, F) \) for the rest of the chapter.

As mentioned above, we specify a model as a collection of processes. To capture this in the semantics we assume that a finite sequence of processes \( P_1, P_2, \ldots, P_N, \) \( N \geq 1 \), is associated with the LTS. We introduce the mapping \( \text{Pid} : R \to \{1, \ldots, N\} \), which assigns to each transition a process index. Intuitively, \( \text{Pid}(s \xrightarrow{a} s') = i \) means that the transition \( s \xrightarrow{a} s' \) is generated by some statement executed by the process \( P_i \). An action \( a \) is enabled in a state \( s \in S \) iff \( s \xrightarrow{a} s' \in R \) for some \( s' \in S \). We say that the process \( P_i \) is enabled in \( s \in S \) iff there exists \( s' \), such that \( s \xrightarrow{a} s' \in R \) and \( \text{Pid}(s \xrightarrow{a} s') = i \). An execution sequence or path is a finite or infinite sequence of subsequent transitions, i.e., for \( s_i \in S, a_i \in A \), the sequence \( s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots \) is an execution sequence in \( T \) iff \( s_i \xrightarrow{a_i} s_{i+1} \in R \) for all \( i \geq 0 \). An infinite execution sequence is said to be accepting iff there is an acceptance state \( s \in F \) that occurs infinitely many times in the sequence. A finite execution sequence \( c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n, n \geq 1 \) is a cycle iff the start and end states coincide, i.e. \( s_0 = s_n \). Given a finite or infinite execution sequence \( \sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots \), a process \( P_i, 1 \leq i \leq N \), and a state \( s_j \) from the execution sequence, we say that \( P_i \) is executed in \( \sigma \) in \( s_j \) iff \( \text{Pid}(s_j \xrightarrow{a_j} s_{j+1}) = i \). A state \( s \) is reachable iff there exists a finite execution sequence that starts at \( \hat{s} \) and ends in \( s \). A cycle \( c \) is reachable iff there exists a state in \( c \) which is reachable. A cycle \( c \) is an acceptance cycle if it contains at least one acceptance state.

Next, we define bisimulation between to LTSs:

Definition 2. Given two LTSs \( T_1 = (S_1, R_1, L_1, A, \hat{s}_1, F_1) \) and \( T_2 = (S_2, R_2, L_2, A, \hat{s}_2, F_2) \), an equivalence relation \( B \subseteq S_1 \times S_2 \) is called a bisimulation between \( T_1 \) and \( T_2 \) iff the following conditions hold:

- \( \hat{s}_1 B \hat{s}_2 \);
- If \( s B s' \), then:
  - \( L_1(s) = L_2(s') \);
  - \( s \in F_1 \iff s' \in F_2 \);
  - Given an arbitrary transition \( s \xrightarrow{a} s_1 \in R_1 \), there exists \( s_2 \in S_2 \) such that \( s' \xrightarrow{a} s_2 \in R_2 \) and \( s_1 B s_2 \);
  - The symmetric condition holds: Given an arbitrary transition \( s' \xrightarrow{a} s_2 \in R_2 \), there exists \( s_1 \in S_1 \) such that \( s \xrightarrow{a} s_1 \in R_1 \) and \( s_1 B s_2 \);
We say that $T_1$ and $T_2$ are bisimilar iff there exists a bisimulation between $T_1$ and $T_2$.

2.2 The Standard Nested Depth-First Search Algorithm

The algorithms presented in this chapter are based on the algorithm of [7] for memory efficient verification of LTL [9] properties, called nested depth-first search (NDFS) algorithm. The algorithm is implemented in the model-checker Spin [15]. In the rest of this section we give a brief overview of the NDFS algorithm.

We begin by considering the basic depth-first search algorithm in Fig. 1. When it is started in the initial state $\hat{s}$ of a given LTS $T$, the basic depth-first search algorithm generates and explores the reachable part of $T$, i.e., all reachable states and all transitions between them.

1 proc dfs1(s)
2 add s to Stack
3 add s to States
4 for each transition $(s,a,s')$
5 add $\{s,a,s'\}$ to Transitions
6 if $s'$ not in States then dfs1(s') fi
7 od
8 delete s from Stack
9 end

Fig. 1. Basic DFS algorithm.

Note that in the algorithm in Fig. 1, as well as in all other algorithms in this chapter, we need to represent only the states of the LTS; the representation of the transitions is not needed. Another remark is that also no explicit representation of the depth-first search stack is needed. Thus, the variables $Transitions$, $Stack$, and the statements from lines 2, 5, and 8 are only used in the presentation and in some of the proofs of the algorithms.

The basic DFS cannot detect cycles. Therefore, in order to do model checking we extend it with a call to a procedure that checks for a cycle, as soon as an acceptance state is encountered. The new algorithm is given in Fig. 2.

The cycle check procedure is again a DFS which reports a cycle and stops the algorithm if the seed acceptance state is matched. If a cycle through the acceptance state does not exist, then the basic DFS is resumed at the point in which it was interrupted by the nested cycle check. In the sequel we also call the basic DFS first DFS, while the nested cycle checks all together are called second DFS. We need to work with two distinct copies of the state space in order to ensure that the second DFS does not fail to detect a cycle by cutting the search because it has hit a state already visited by the first DFS.
1 proc dfs1(s)
2   add s to Stack1
3   add s to States1
4   if accepting(s) then States2:=empty; seed:=s; dfs2(s) fi
5   for each transition (s,a,s’) do
6     add {s,a,s’} to Transitions
7     if s’ not in States1 then dfs1(s’) fi
8   end
9 delete s from Stack1
10 end

11 proc dfs2(s) /* the nested search */
12   add s to Stack2
13   add s to States2
14   for each transition (s,a,s’) do
15     add {s,a,s’} to Transitions2
16     if s’ == seed then report cycle
17     else if s’ not in States2 then dfs2(s’) fi
18   od
19 delete s from Stack2
20 end

Fig. 2. Nested depth first search (NDFS) algorithm, version 1 (“preorder”).

An important feature of the algorithm in Fig. 2 is that before the cycle check is called its copy of the state space is reset in line 4 with the statement States2:=empty. Hence, the cycle check is always started from scratch. As a consequence, some states can be visited several times (by different cycle checks), which increases the time complexity of the algorithm. The reinitialization of States2 is needed, because otherwise some acceptance cycles can be missed. An example of an LTS for which the algorithm without reinitialization fails is shown in Fig. 3. The states \( \hat{s} \) and \( s_1 \) are acceptance states. All the states are visited and

\[ \hat{s} \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \]

Fig. 3. A counterexample LTS for the NDFS with state space preservation.

put in States2 during the first cycle check which is started from \( \hat{s} \). Thus, when the second cycle check is called from \( s_1 \), it is stopped at \( s_2 \) which was entered in
States2 by the previous cycle check. As a result, the cycle $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_1$ is not reported.

In order to be able to preserve the States2 between cycle checks and reuse the previous calls of dfs2, we need a small modification of the algorithm—the cycle check should start only after all the successors of an acceptance state are explored, i.e., when the recursion retracts from an acceptance state [7]. This is achieved by moving line 4 to the end of the procedure dfs1, i.e., after line 8, and removing the statement $\text{States2} := \text{empty}$ which is not needed anymore.

Also, in the new version of the algorithm we use only one $\text{States}$ (and Transitions) variable and extend the state representation with an additional bit to distinguish between the two different copies of the state space, belonging to the first and second DFS, respectively. The variable $\text{States}$ is preserved between the calls of dfs1 and dfs2 and, as a result, each state is visited only once also during the second DFS. The resulting algorithm is shown in Fig. 4.

```
1 proc dfs1(s)
2    add $s$ to Stack1
3    add {$s$,$0$} to States
4    for each transition $(s,a,s')$ do
5        add {$s$,$0$},a,{$s'$,$0$} to Transitions
6        if {$s'$,$0$} not in States then dfs1(s') fi
7    od
8    if accepting(s) then seed:={$s$,$1$}; dfs2(s) fi
9    delete s from Stack1
10 end

11 proc dfs2(s) /* the nested search */
12    add $s$ to Stack2
13    add {$s$,$1$} to States
14    for each transition $(s,a,s')$ do
15        add {$s$,$1$},a,{$s'$,$1$} to Transitions
16        if {$s'$,$1$} == seed then report cycle
17        else if {$s'$,$1$} not in States then dfs2(s') fi
18    od
19    delete s from Stack2
20 end
```

Fig. 4. Nested depth first search (NDFS) algorithm.

The following claim (Theorem 1 from [7]) establishes the correctness of the algorithm

**Theorem 1 ( [7]).** Given an LTS $T$, the NDFS algorithm in Fig. 4, when called on $s$, reports a cycle iff there is a reachable acceptance cycle in $T$. 

The cycle which is reported is contained in Stack2, while Stack1 contains the path from the initial state that leads to it. Thus, the counterexample execution can be reproduced by concatenating the contents of the stacks.

The fact that for each state \( s \) the copies in the first and second DFS differ only in the second (bit) component can be used to save memory space [18]. The states \( (s, 0) \) and \( (s, 1) \) can be stored together as \( (s, b_1, b_2) \), where \( b_1 \) (respectively \( b_2 \)) is a bit which is set to 1 iff \( (s, 0) \) (resp. \( (s, 1) \)) has been already visited during the first (resp. second) DFS.

The NDFS algorithm can be optimized by reporting a cycle when a state is visited which is already on the stack of the first DFS [18]. Unfortunately, this optimization does not work when NDFS is combined with partial order reduction [18].

3 Weak Fairness

We consider weak fairness with regard to processes, i.e. we say that a given execution sequence is fair if for each process that becomes continuously enabled starting at some point in the execution sequence, a transition belonging to this process is executed infinitely many times. Formally:

**Definition 3.** An infinite execution sequence \( s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots \) is fair iff for each process \( P_l, 1 \leq l \leq N \) the following holds: If there exists \( i \geq 0 \) such that \( P_l \) is enabled in \( s_j \) for all \( j \geq i \), then there are infinitely many \( k \geq 0 \) such that \( P_l \) is executed in \( s_k \).

A cycle \( c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0 \) is fair iff whenever a process \( P \) is enabled in all states \( s_i, 0 \leq i < n \), then \( P \) is executed in some state \( s_j, 0 \leq j < n \).

When solving the model-checking problem under the (weak) fairness assumption we are interested only in fair accepting execution sequences. As we work with finite LTSs, it is obvious that translated in terms of acceptance cycles the fairness assumption means that we require that the acceptance cycles we detect in the state space are fair.

3.1 Description of the Weak Fairness Algorithm

The weak fairness (WF) algorithm presented in the sequel is a modification of the algorithm implemented in the model checker Spin by Gerard Holzmann. It is a variant of Choueka’s flag algorithm [16].

Consider the following straightforward adaptation of the standard NDFS:

Each time the second DFS is resumed from a seed (acceptance) state, keep track which processes have contributed a transition or have been disabled so far, starting from the seed state. Whenever the seed state is matched, i.e., an acceptance cycle is detected, check if all the processes have contributed a transition or have been disabled in some state. If this is not true, then the cycle is not fair and simply ignore it, backtrack and
continue the search until a fair cycle is found or all acceptance states are checked without detecting a cycle.

This approach does not work because the NDFS algorithm guarantees to report at least one acceptance cycle (if there are any), but not all of them. Consequently, fair acceptance cycles could remain undetected.

To bypass the above described limitations of the NDFS, we apply it not to the original state space, but to an extended state space. The latter is constructed such that the existence of a fair acceptance cycle in the original state space implies that there exists an acceptance cycle in the extended state space, and vice versa.

We first give the intuition behind the extended state space and the modified NDFS algorithm. The set of states of the extended state space consists of $N$ copies of the original state set, where $N$ is the number of processes. The intuition behind the copies is that when the algorithm works in the $i$-th copy of the state space, this means that it waits for process $i$ to contribute a transition or to become disabled. (In order to treat these two cases uniformly, we will assume that when process $i$ is disabled in the original state space, in the extended state space it executes a *default transition*, labeled with $\epsilon_i$.) When the algorithm is in copy $i$ and process $i$ executes a transition, then the algorithm passes to copy $i + 1$. From copy $N$ it goes back to copy 1. Let us suppose that the acceptance states of the extended system are the acceptance states of the original system which reside in copy 1, and that the cycle check always starts in copy 1. Obviously, if a cycle goes through all $N$ copies and arrives back to copy 1, then it is fair. Unfortunately, cycles through the acceptance states can be closed without leaving copy 1. A remedy for this problem is to ensure that from an acceptance state we can only go to a state which is in the next copy. Apparently, copy 1 (or any other copy) cannot in general satisfy this requirement because it is in conflict with the requirement that we pass to copy $i + 1$ only by taking transitions of process $i$.

Therefore, we introduce a new special copy 0, which does not correspond to any process and to which we move the acceptance states. Copy 0 is inserted between copy $N$ and copy 1, i.e., now we pass from copy $N$ to copy 0, instead of copy 1. Whenever we are in an acceptance state in copy 0 we go via a special $\tau$-transition to copy 1. The $\tau$-transitions do not belong to any process and do not correspond to any transition from the original system. Note that the only possible transitions from an acceptance state of the extended state space are $\tau$-transitions. We stay inside copy 0 as long as transitions originating from a non-acceptance state are explored. The extended state space with copy 0 and its relation to the original state space is shown in Figure 5.

$T$ and $F$ denote the original state space and its acceptance states, respectively. The extended state space is denoted with $F(T)$, the set of its acceptance states with $F_f$, while $T_i, 1 \leq i \leq N$ are the copies of the original state space $T$. With $T_0 - F_f$ we denote copy 0 of the original state space without the acceptance states $F_f$. The label $\text{Pid}(t) = i$ ($\text{Pid}(t) \neq i$) on the arrows between the copies express the fact that we can pass between the copies $i$ and $i + 1$ (resp. stay inside copy $i$) by executing a transition belonging to process $i$. One can see that in the
new setting all the cycles that pass through an acceptance state must also pass through all the copies and, consequently, they are all fair.

For representing the extended state space, we extend each original state with an integer counter that denotes the current copy of the state set. Formally, the extended state space is defined as follows:

**Definition 4.** Given an LTS $T = (S, R, L, A, \hat{s}, F)$ and processes $P_1, \ldots, P_N$, $N \geq 2$, we define its (weakly) fair extension to be the LTS $F(T) = (S_f, R_f, L_f, A \cup \{\tau, \epsilon_1, \ldots, \epsilon_N\}, \hat{s}_f, F_f)$ as

- $S_f = S \times \{0, 1, \ldots, N\}$
- $R_f$ is defined with the following cases:
  1. $(s, 0) \xrightarrow{a} (s', 0) \in R_f$ iff $s \not\in F$ and $s \xrightarrow{a} s' \in R$;
  2. $(s, 0) \xrightarrow{\tau} (s, 1) \in R_f$ iff $s \in F$;
  3. $(s, C) \xrightarrow{a} (s', C) \in R_f$ iff $C > 0$ and $s \xrightarrow{a} s' \in R$ and $\text{Pid}(s \xrightarrow{a} s') \neq C$;
  4. $(s, C) \xrightarrow{a} (s', (C + 1) \mod (N + 1)) \in R_f$ iff $C > 0$ and $s \xrightarrow{a} s' \in R$ and $\text{Pid}(s \xrightarrow{a} s') = C$;
  5. $(s, C) \xrightarrow{C} (s, (C + 1) \mod (N + 1)) \in R_f$ iff $C > 0$ and $PC$ is disabled in $s$;
- for all $(s, C) \in S_f, L_f((s, C)) = L(s)$.
- $\hat{s}_f = (\hat{s}, 0)$
- $F_f = \{(s, 0) \mid s \in F\}$.

![Fig. 5. The extended state space for the weak fairness algorithm.](image-url)
$\text{Pid}((s,C) \xrightarrow{\tau} (s',C')) = 0$, $\text{Pid}((s,C) \xrightarrow{\epsilon} (s',C')) = i$, and $\text{Pid}((s,C) \xrightarrow{a} (s',C')) = \text{Pid}(s \xrightarrow{a} s')$, for all $1 \leq i \leq N$, $(s,C),(s',C') \in S_f$ and $a \in A$.

The most direct way to do the model checking under weak fairness is to first unfold the original state space $T$ into its weakly fair extension $F(T)$ and then to apply the NDFS algorithm on $F(T)$. Instead, for efficiency reasons, we do the unfolding of the state space and the NDFS simultaneously. In this way we keep the advantage of the on-the-fly approach that, if there exists a fair acceptance cycle, usually we do not have to generate the whole $F(T)$.

The merge of the two stages is straightforward. The pseudo-code of the algorithm is given in Figure 6. The code is basically the standard NDFS model-checking algorithm modified for manipulation of the new counter component in order to generate the extended state space according to Definition 4.

We represent the states of the extended state space as triples of the form $(s,C,b)$. They are obtained in a straightforward way by extending each state $s$, apart from the bit $b$ discriminating between the first and second DFS, also with the integer counter $C$ that keeps track of the current copy of the original state space.

The line

```
for each transition (s,a,s') do
```

that occurs in the standard NDFS algorithm in Fig. 4 in the WF algorithm in Fig. 6 is expanded into

```
for each process i = 1 to N do
    ...
    nxt = all transitions enabled in s with Pid(t)=i
    ...
    for all (s,a,s') in nxt do
```

This refinement is necessary because in the WF algorithm the process identities play a significant role.

The other modifications that are added to the standard NDFS scheme are because of Definition 4, more precisely, to capture the transition relation $R_f$. The $\tau$-transitions (case 2 of the definition) are generated with the true branch of the outermost if (lines 4-6 and 28-30). The increment of the counter component by 0 or 1 in the cases 3 to 5 is reflected in the line 9 and 33. The $\epsilon$-transitions (case 5) are implemented with the true branch of the if statement (line 11-13 and 35-38). Finally, case 0 is also implicitly implemented through line 9 and 33. As variable $i$ is never 0, if $C$ is 0, then the assignment $C' = C$ is executed, which leaves the counter component unchanged.

### 3.2 Correctness of the Weak Fairness Algorithm

We first show that the WF algorithm is in fact the NDFS algorithm applied to the fair extension $F(T)$. The main idea is to execute in lock-step the two algorithms and show that they produce the same state space.
proc dfs1(s,C)
  add {s,C} to Stack1
  add {s,C,0} to States
  if C == 0 and accepting(s) then
    add {{s,0,0},tau,{s,1,0}} to Transitions
    if {s,1,0} not in States then dfs1(s,1) fi /* tau move */
  else
    for each process i := 1 to N do
      if C == i then C' := (C+1) mod (N+1) else C' := C fi
      nxt := all transitions t enabled in s with Pid(t) == i
      if nxt == empty then /* epsilon move */
        if C == i then add {{s,C,0},epsilon,{s,C',0}} to Transitions fi
        if {s,C',0} not in States and C == i then dfs1(s,C') fi
      else
        for all (s,a,s') in nxt do
          add {{s,C,0},a,{s',C',0}} to Transitions
          if {s',C',0} not in States then dfs1(s',C') fi
        od
      fi
    od
    if C == 0 and accepting(s) then seed := {s,C,1}; dfs2(s,C) fi
  delete {s,C} from Stack1
end

proc dfs2(s,C) /* the nested search */
  add {s,C} to Stack2
  add {s,C,1} to States
  if C == 0 and accepting(s) then
    add {{s,0,1},tau,{s,1,1}} to Transitions
    if {s,1,1} not in States then dfs2(s,1) fi /* tau move */
  else
    for each process i := 1 to N do
      if C == i then C' := (C+1) mod (N+1) else C' := C fi
      nxt := all transitions t enabled in s with Pid(t) == i
      if nxt == empty then /* epsilon move */
        if C == i then add {{s,C,1},epsilon,{s,C',1}} to Transitions fi
        if {s,C',1} == seed then report cycle
        else if {s,C',1} not in States and C == i then dfs2(s,C') fi
      else
        for all (s,a,s') in nxt do
          add {{s,C,1},a,{s',C',1}} to Transitions
          if {s',C',1} == seed then report cycle
          else if {s',C',1} not in States then dfs2(s',C') fi
        od
      fi
    od
    if C == 0 and accepting(s) then delete {s,C} from Stack2
end

Fig. 6. Weak fairness (WF) algorithm.
Lemma 1. Given an LTS $T$, for every execution $E$ of the NDFS algorithm from Fig. 4, started in $(\hat{s}, 0) \in F(T)$, there exists an execution $E'$ of the WF algorithm, started in $\hat{s} \in T$, such that the parts of $F(T)$ which are generated and explored by $E$ and $E'$ are the same.

Proof. We will construct an execution $E'$ of the WF algorithm applied to $T$ while tracing the execution $E$ applied to $F(T)$. We denote the $j$-th element from the bottom of the $Stack_i, i = 1, 2$, with $Stack_i(j)$, i.e., $Stack_i(0)$ is the bottom element. The function $\text{length}(Stack_i)$ returns the number of elements in $Stack_i$. Thus $Stack_i(\text{length}(Stack_i) - 1)$ is the top element. The superscript of a variable denotes the execution, i.e., algorithm, it belongs to.

We will show that at each point the following invariants hold:

- $\text{length}(Stack^E_i) = \text{length}(Stack^{E'}_i), i = 1, 2$, and $Stack^E_i(j) = Stack^{E'}_i(j), i = 1, 2, 0 \leq j \leq \text{length}(Stack^{E'}_i) - 1$, and
- $\text{States}^E_i = \text{States}^{E'}_i$ and $\text{Transitions}^E_i = \text{Transitions}^{E'}_i$.

Initially, the invariants hold because both $E$ and $E'$ begin by adding $(\hat{s}, 0)$ to $\text{States}$ and $\text{Stack}$. Now, we advance $E$ and show how it can be mimicked by $E'$, while preserving the invariants. Suppose that at some point in $E$ a $\tau$-transition is generated and added to the state space. From the Def. 4 it follows that the state which is on the top of the $Stack_i$ and which is currently visited in by the NDFS, i.e., in the execution $E$, is an acceptance state. By the invariant, the same state is also on the top of the $Stack_{E'}$. Thus, as the counter component of this state is 0, because of the lines 4-6 and 28-30 in $dfs1$ and $dfs2$, respectively, the same $\tau$-transition is added to the $\text{States}_{E'}$. This is because the state which is generated by the $\tau$-transition is the same in both executions. Also, as the $\text{States}$ variables contain the same states, the search is continued via a recursive call to $dfs1$ ($dfs2$) in NDFS ($E$) if and only if the same is done in WF ($E'$). Moreover, in both algorithms the same state is passed as an argument to the called procedure, which implies that the same state will be pushed in their $\text{Stack}$ variables by both algorithms.

In a similar way one can show using Def. 4 that one can mimic in $E'$ the $\epsilon$- (lines 10-12 and 35-37), as well as the ordinary transitions (lines 14-16 and 39-42). We assume that in the for each statements of NDFS the transitions from $s$ are taken in the same order as in the nested loops (for each and for all) in the WF algorithm. In other words, the processes are checked for enabled transitions in increasing order, starting at process 1, and the transitions of each process are chosen in the same order as in the statement

for all $(s,a,s')$ in nxt do

in lines 14 and 39 of WF. \qed

It is worth noting that $C = k$, for some $1 \leq k \leq N$, implies that the processes with indices less then $k$ have been executed or been disabled in some state after the last “restarting” of the counting via a $\tau$-transition from some acceptance state.
The next step is to prove the correspondence between the fair acceptance cycles in the original state space and the acceptance cycles in its weakly fair extension.

**Lemma 2.** There exists a reachable fair acceptance cycle in $T$ iff there exists a reachable acceptance cycle in $F(T)$.

**Proof.** Let $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ be a reachable fair acceptance cycle in $T$. Let us suppose without loss of generality that there is a path $p$ from the initial state $\hat{s}$ to $s_0$. By applying the definition of $R_f$ (Def. 4) $p$ can be mapped into a path $p'$ from $(\hat{s}, 0)$ to $(s_0, C)$, for some $C$.

We can continue extending $p'$ by mapping $c$, starting from $s_0$ and according to the definition of $R_f$. (Note that we might have to pass the cycle $c$ several times.) The only ambiguity that can occur during the mapping is resolved such that the $\epsilon$-transitions (i.e. Case 5 in Def. 4) have priority over the ordinary transitions (i.e. Case 3 in Def. 4). More precisely, when we are extending from some state $(s, C)$ and the corresponding transition in $c$ is $s \xrightarrow{a} s'$, then if $P_C$ is disabled in $s$, the path in $F(T)$ is extended with $(s, C) \xrightarrow{\tau} (s, (C + 1) \text{ mod } (N + 1))$, otherwise the extension is $(s, C) \xrightarrow{a} (s', C)$. As $c$ is fair and contains acceptance states we will arrive eventually in $F(T)$ in some acceptance state $(s_a, 0)$. (The number of steps can be zero, if $(s_0, C)$ is already an acceptance state.) If we keep cycling and mapping along $c$, again because of the fact that $c$ is a fair acceptance cycle, we will come again across an acceptance state $(s_a, 0)$. If $s_{a0} = s_{a1}$ we are done because we have found the desired acceptance cycle in $F(T)$. Otherwise, we repeat the same procedure starting from $(s_{a1}, 0)$. Because the number of acceptance states along $c$ is finite we will eventually end up in $F(T)$ in some acceptance state $(s_{a_k}, 0)$ that we have already visited. It is obvious from the above construction that the obtained acceptance cycle passing through $(s_{a_k}, 0)$ is reachable from the initial state $(\hat{s}, 0)$.

For the reverse direction, we show that each acceptance cycle $c$ from $F(T)$ is projected into a fair acceptance cycle $c'$ in $T$. The last property follows directly from the definition of $F(T)$. Each acceptance cycle in $F(T)$ contains a $\tau$-transition from some acceptance state $(s, 0)$ to the state $(s, 1)$. This transition is the only one from $(s, 0)$ and the counter component along the cycle can be changed only by increasing it modulo $N + 1$. Thus, in order to get back to $(s, 0)$ each process must execute a transition along the cycle (ordinary or $\epsilon$). The desired fair acceptance cycle in $T$ is obtained by simply omitting the counter component of the states in $c$ and eliminating the $\epsilon$- and $\tau$-transitions by merging the states which are connected via these. As $c$ is reachable in $F(T)$, some $(s, C)$ on $c$ is also reachable via some path $p$ in $F(T)$ which starts at the initial state $(\hat{s}, 0)$. The path $p$ too can be projected by omitting the counter components and the $\epsilon$-transitions into a path $p'$ in $T$ that leads to the state $s$ which is on $c'$.

Finally, Lemma 1, Lemma 2 and Theorem 1 put together give the correctness of the WF algorithm:
Theorem 2. Given an LTS $T$, the weak fairness algorithm (WF) from Figure 6, when called on $(\hat{s}, 0)$, reports a cycle if and only if there exists a reachable fair acceptance cycle in $T$.

4 Combining Symmetry Reduction and Weak Fairness

In this section we combine the ideas of the weak fairness algorithm from the previous section with reduction techniques that exploit symmetry. We first treat the simpler case of a NDFS algorithm for symmetry reduction without fairness. Using the approach of [12] we present this algorithm in the framework of bisimulation preserving reduction, which is a generalization of the symmetry reduction.

It should be emphasized once again that we assume that we work on a state space which is the product of the system model and the property, i.e., a Büchi automaton. In order to ensure bisimulation preservation, both the model and the property (Büchi automaton) have to satisfy certain conditions. Most of the time these conditions are just sufficient conditions which are efficiently checkable in practice, preferably on syntactic level [19, 10, 11]. In what follows we abstract from them and assume that they are captured in the selection function which we define below.

4.1 Bisimulation Preserving Reduction without Fairness

We begin by recalling some definitions and results from [12].

The main idea is to perform the model checking on an abstract state space, which is usually much smaller than the original one. To this end, the original state set $S$ is partitioned into equivalence classes, by means of some function $h : S \rightarrow S$, such that two states $s_1$ and $s_2$ are in the same class iff $h(s_1) = h(s_2)$. The abstract state space consists of the representatives of these classes with transitions between them as defined below.

Definition 5. Given a function $h : S \rightarrow S$ on LTS $T = (S, R, L, A, \hat{s}, F)$, we define the corresponding abstract LTS $h(T)$ to be $(S_h, R_h, L_h, A, h(\hat{s}), F_h)$, where

- $S_h = h(S)$, the set of representatives,
- $r_1 \xrightarrow{a} r_2 \in R_h$ with $\text{Pid}(r_1 \xrightarrow{a} r_2) = i$ iff there exists $s \in S$ such that $r_1 \xrightarrow{a} s \in R$ with $\text{Pid}(r_1 \xrightarrow{a} s) = i$ and $h(s) = r_2$.
- for all $r \in S_h$, $L_h(r) = L(r)$, and
- $F_h = h(F)$.

Usually, several different transitions from $T$ are represented by the same transition in $h(T)$. As a result, only the pids of the representative transitions are preserved, while those of the other transitions are lost by the reduction.

In order to obtain a correspondence between the acceptance cycles in $T$ and $h(T)$ we need to impose some additional constraints on the function $h$.

Definition 6. For a given LTS $T$, a function $h : S \rightarrow S$ is a selection function iff there exists a bisimulation $B \subseteq S \times S$ between $T$ and $T$ such that
1. for all \( s \in S \), \( sBh(s) \).
2. \( sB s' \) implies that \( h(s) = h(s') \).

In the sequel we assume that \( h \) is a selection function. We say that \( h \) preserves the bisimulation relation \( B \). Intuitively, the function \( h \) picks a representative for each equivalence class of \( S \) induced by \( B \).

The following result (Lemma 8 from [12]) is implied directly by the definitions given above:

**Lemma 3 ([12]).** Given an LTS \( T = (S, R, L, A, \hat{s}, F) \) and a selection function \( h, T \) and \( h(T) \) are bisimilar.

As a consequence we can do the model checking in the reduced state space. In the sequel we show how this can be done with a variation of the NDFS algorithm applied to the reduced state space \( h(T) \). We begin with the following claim:

**Lemma 4.** Given an LTS \( T \) and a selection function \( h : S \rightarrow S \) for \( T \)

1. for each path \( p = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n \) in \( T \), there exists a corresponding path \( q = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} \ldots r_{n-1} \xrightarrow{a_{n-1}} r_n \) in \( h(T) \), such that \( r_i = h(s_i), 0 \leq i \leq n \).
2. if \( h \) is a selection function, then for each path \( p = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} \ldots r_{n-1} \xrightarrow{a_{n-1}} r_n \) in \( h(T) \) and every state \( s_0 \) such that \( h(s_0) = r_0 \), there exists a corresponding path \( q = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n \) in \( T \), such that \( r_i = h(s_i), 0 \leq i \leq n \).

**Proof.**
1. From \( r_0 = h(s_0) \) by point 1 of Def. 6 we conclude that \( s_0B r_0 \). Thus, there exists \( s'_1 \in S \) such that \( r_0 \xrightarrow{a_0} s'_1 \in R \) in \( T \) and \( s'_1B s_1 \). By point 2 of Def. 6 \( h(s'_1) = h(s_1) \). Thus, by Def. 5 there exists in \( h(T) \) a transition \( r_0 \xrightarrow{a_1} r_1 \), where \( r_1 = h(s_1) \). Using analogous arguments we can show by induction on \( i \) that for each transition \( s_i \xrightarrow{a_i} s_{i+1} \in R \) from \( p \) there exists a corresponding transition \( r_i \xrightarrow{a_i} r_{i+1} \in R_h \) from \( q \), \( 1 \leq i \leq n - 1 \), which proves the existence of \( q \).
2. By point 1 of Def. 6 \( h(s_0) = r_0 \) implies \( r_0B s_0 \). Since \( r_0 \xrightarrow{a_0} r_1 \), there exists \( s_1 \) such that \( s_0 \xrightarrow{a_0} s_1 \) and \( r_1B s_1 \). Applying point 2 of Def. 6 we obtain that \( h(s_1) = h(r_1) \). Thus, we need to show that \( h(r_1) = r_1 \). By point 1 of Def. 6 we have that \( r_1B h(r_1) \). Now we use the fact that \( r_1 \) is a unique representative in \( h(T) \) of its equivalence class (under \( B \)). More precisely, from the definition of \( h(T) \) (Def. 5) it follows that there exists a state \( s \) in \( T \) such that \( h(s) = r_1 \). By point 1 of Def. 6 \( sB r_1 \). Using again point 2 of Def. 6 one obtains \( h(s) = h(r_1) \), and consequently, \( h(r_1) = r_1 \). Continuing in this way, we construct the path \( q \) by finding for each transition \( r_i \xrightarrow{a_i} r_{i+1} \in R_h \) from \( p \), a corresponding transition \( s_i \xrightarrow{a_i} s_{i+1} \in R, 1 \leq i \leq n - 1 \).

\( \Box \)

**Lemma 5.** There exists a reachable acceptance cycle in the LTS \( T \) iff there exists a reachable acceptance cycle in \( h(T) \).
Proof. Let \( c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0 \) be a reachable acceptance cycle in \( T \) and let us suppose without loss of generality that \( s_0 \) is an acceptance state. Because \( c \) is reachable, also any state on \( c \), and therefore \( s_0 \), is reachable in \( T \).

Thus, there exists in \( T \) a path \( p = \hat{s} \xrightarrow{b_0} q_1 \xrightarrow{b_1} \ldots q_{m-1} \xrightarrow{b_{m-1}} s_0 \). By point 1 of Lemma 4 there exists a path from \( h(\hat{s}) \) to \( h(s_0) \), and therefore \( h(s_0) \) is reachable in \( h(T) \).

Similarly, point 1 of Lemma 4 implies that \( c' = h(s_0) \xrightarrow{a_0} h(s_1) \xrightarrow{a_1} \ldots h(s_{n-1}) \xrightarrow{a_{n-1}} h(s_0) \) is a cycle in \( h(T) \). As by Def. 5 \( h(s_0) \) is an acceptance state, it follows that \( c' \) is an acceptance cycle in \( h(T) \).

For the reverse direction, let \( c = r_0 \xrightarrow{a_0} r_1 \xrightarrow{a_1} \ldots r_{n-1} \xrightarrow{a_{n-1}} r_n \), where \( r_0 = r_n \), be a reachable acceptance cycle in \( h(T) \). Like in the previous cases, without loss of generality let us assume that \( r_0 \) is an acceptance state.

By point 2 of Lemma 4 \( c \) can be mapped into a path \( p = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n \) in \( T \), such that \( r_i = h(s_i), 1 \leq i \leq n \). If \( s_0 \neq s_n \), i.e., the path \( p \) is not a cycle, we repeat the mapping of \( c \), starting this time from \( s_n \). We continue this unfolding of \( c \) in \( T \) until we close a cycle \( c' \) in \( T \). Obviously, this will eventually happen, because the number of times we repeat the mapping of \( c \) is limited by the number of states in the equivalence class of \( s_0 \) (which is finite because there are only finitely many states in an LTS). Moreover, \( c' \) contains \( s_0 \) or some other state from its equivalence class, therefore, an acceptance state.

It remains to prove that \( c' \) is reachable in \( T \). By point 2 of Lemma 4 the above described construction of \( c' \) can be started in any state \( s_0 \), such that \( r_0 = h(s_0) \).

We show that at least one such a state (and therefore \( c' \)) which is constructed starting from it) is reachable in \( T \). Because \( c \) is reachable, there exists a path in \( h(T) \) from the initial state \( h(\hat{s}) \) to \( r_0 \). By point 2 of Lemma 4 there exists a corresponding path in \( T \) from \( \hat{s} \) to some state \( s_0 \) such that \( r_0 = h(s_0) \). Thus, \( s_0 \) is reachable in \( T \).

We generate and search the reduced abstract state space for acceptance cycles with a straightforward modification of the standard (postorder) NDFS from Figure 4, which we call reduced NDFS (RNDFS). To this end the searched state space is made to comply with Def. 5. More precisely, RNDFS is obtained from the standard NDFS algorithm by replacing all the occurrences (except in lines 4 and 14) of the newly generated state \( s' \) with its representative \( h(s') \). The RNDFS algorithm is given in Fig. 7.

As the RNDFS algorithm is in fact NDFS applied to the reduced state space, its correctness follows directly from the correctness of the NDFS algorithm (Theorem 1) and the bidirectional cycle correspondence Lemma 5. Thus, we have the following claim:

**Theorem 3.** Given an LTS \( T \) and a selection function \( h \), the nested depth first search algorithm with bisimulation preserving reduction (RNDFS) in Figure 7, when called on (the initial state) \( h(\hat{s}) \), reports a cycle if and only if there is a reachable acceptance cycle in \( T \).

An important advantage of the standard NDFS algorithm is that the erroneous execution can be recreated by dumping the contents of the stack (Stack1...
Fig. 7. Nested depth first search algorithm with bisimulation preserving reduction (RNDFS).

and Stack2. Unfortunately, this is no longer true with RNDFS. The reason is that in RNDFS \texttt{dfs1} and \texttt{dfs2} are not called with the newly generated states as arguments, but instead from their representatives (lines 6 and 16). Consequently, the stacks might contain a sequence of states which corresponds to an execution sequence that do not exist in the original LTS $T$. (This effect of introducing transitions between representative states in $h(T)$ is obvious from Def. 5.) In order to solve this problem, we modify the RNDFS algorithm such that \texttt{dfs1} and \texttt{dfs2} are called on the newly generated states and also the original states are saved on the stack instead of their representatives. However, in order to still benefit from the reduction only the representatives are saved in \texttt{States}, \texttt{Transitions} and the \texttt{seed}. The modified RNDFS (MRNDFS) is given in Fig. 8.

Notice that in fact the MRNDFS algorithm explores part of the original state space $T$, i.e., part of the original execution sequences, while building the abstract state space. As argued above, this is because always the original state is unfolded in the line

\begin{verbatim}
for each transition (s,a,s') do
\end{verbatim}

and also the procedures \texttt{dfs1} and \texttt{dfs2} are called with the original state as argument, i.e. the original state is stored in \texttt{Stack1} or \texttt{Stack2}. The exploration of the original state space is pruned whenever an equivalent state has already been explored.
Fig. 8. Modified nested depth first search algorithm with bisimulation preserving reduction (MRNDFS).

Next, we show that the algorithms MRNDFS and RNDFS are equivalent in the sense that they produce (part of) the same LTS $h(T)$.

**Lemma 6.** Given an LTS $T$ and a selection function $h$, for every execution $E$ of the RNDFS algorithm on $h(T)$, started in $h(\hat{s})$, there exists an execution $E'$ of the MRNDFS algorithm on $T$, started in $\hat{s}$, such that the parts of $h(T)$ which are saved in the variables $States$ and $Transitions$ in both algorithms are the same.

**Proof.** The proof is similar to the proof of Lemma 1. We construct an execution $E'$ of the MRNDFS algorithm applied to $T$ while tracing the execution $E$ of RNDFS applied to $h(T)$. Using the same denotations as in the proof of Lemma 1, we define analogous invariants, i.e., we show that at each point the following holds:

1. $\text{length}(\text{Stack}^E_i) = \text{length}(\text{Stack}^{E'}_i), i = 1, 2,$ and $\text{Stack}^{E'}_i(j) = h(\text{Stack}^E_i(j)), i = 1, 2, 0 \leq j \leq \text{length}(\text{Stack}^E_i - 1),$ and
2. $States^E = States^{E'}$ and $Transitions^E = Transitions^{E'}$.

Initially, the invariants hold. This is because the execution $E$ begins by adding both $(h(\hat{s}), 0)$ to $States$ and $Stack$. The execution $E'$ in the very beginning also adds $h(\hat{s})$ to its $States$ after pushing $\hat{s}$ in its $Stack$. We show that the lockstep execution of $E$ and $E'$ preserves the invariants. Let $s_E$ and $s_{E'}$ be the states
which are currently visited by $E$ and $E'$, respectively. The states are the top elements of the corresponding Stack variables, both in $E$ and $E'$. Thus, it follows by the first invariant that $s_E = h(s_{E'})$. This implies by Def. 6 that there exists a bisimulation $B$ such that $s_E B s_{E'}$. Thus, there exists a transition which can be taken in the MRNDFS (i.e. $E'$), and which corresponds to the transition which is taken in the RNDFS (i.e. $E$). When the RNDFS algorithm chooses successor states in line 4, this can be done in any order. We assume that the transitions in the for all statements are chosen such that the following holds: if $s a_1 \rightarrow s_1$ is selected by MRNDFS before $s a_2 \rightarrow s_2$, the transition $h(s) a_1 \rightarrow h(s_1)$ is selected by RNDFS before the transition $h(s) a_2 \rightarrow h(s_2)$. The existence of the last two transitions is implied by the bisimilarity between $s$ and $h(s)$. Because the representatives of each equivalence class are unique, the same transition is added to Transitions in both algorithms. Note that in general MRNDFS can explore a different number of transitions in its for all statement than RNDFS in the same statement. This is because, in general, two bisimilar states do not have the same number of outgoing transitions labeled with the same action and leading to corresponding bisimilar states. However, this is not a problem because the extra transitions do not cause additional states and transitions to be saved in Transitions and States. Thus, they can be executed in the for all iteration, for instance, after all the other transitions are explored, without any effect to Stack, States and Transitions. The obtained successor states are bisimilar too and consequently dfs1 and dfs2 are called in the RNDFS iff they are called in MRNDFS. Moreover the argument states are bisimilar which preserves the elementwise bisimilarity of the Stacks and implies that the same states are added to States when the recursive call is entered.

As by Lemma 6 the MRNDFS algorithm boils down to the RNDFS algorithm (which is applied to the abstract state space $h(T)$), its correctness is a corollary of Theorem 3:

**Theorem 4.** Given an LTS $T$, the modified nested depth first search algorithm with bisimulation preserving reduction (MRNDFS) in Figure 8, when called on $\hat{s}$, reports a cycle if and only if there is a reachable acceptance cycle in $T$.

### 4.2 Bisimulation Preserving Reduction with Weak Fairness

Unlike for the ordinary acceptance cycles, there does not exist a correspondence between the fair acceptance cycles in $T$ and $h(T)$. This can be seen on the following example:

**Example 1.** Consider the LTS in Fig. 9. State $s_0 = \hat{s}$ is the initial state and $F = \{s_1, s_4\}$. For simplicity, the edges of $T$ are labeled only with the transition Pids. Obviously, the acceptance cycle $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_2 \rightarrow s_1$ in $T$ is fair, while

\[\begin{align*}
\text{2} & \text{ However, for symmetry reduction, that we introduce later, this number is always the same, because of the specific way the selection function } h \text{ is chosen.} \\
\text{3} & \text{ We owe this example to Dennis Dams}
\end{align*}\]
the reduced LTS $h(T)$, where $h(s_0) = s_0$, $h(s_1) = s_1$, $h(s_2) = s_2$, $h(s_3) = s_2$, $h(s_4) = s_1$, $h(s_5) = s_5$, and $h(s_6) = s_5$, given in Fig. 10, does not contain a fair acceptance cycle.

From the discussion above it is obvious that the straightforward approach of applying the (weak) fairness algorithm from Section 3 to $h(T)$ is not going to work.

Another straightforward way to model check under weak fairness utilizing the bisimulation preserving reduction is to incorporate the weak fairness requirement into the formula (automaton) that expresses the property. In that way the RNDFS algorithm can be used in its original form from Fig. 7. However, due to the additional conditions imposed on $h$ by the property component
of the states of $T$, this approach can be quite ineffective. For instance, it is known that for the case when the bisimulation is symmetry reduction (that we consider in a moment), such an approach does not give any reduction [11].

However, one can find satisfactory solutions for special cases of bisimulation preserving reductions. In the next section we present one such algorithm for symmetry reduction.

### 4.3 Symmetry Reduction with Weak Fairness

The algorithm for combining symmetry reduction with weak fairness that we present below is based on the theory developed in [11]. Here we do the necessary adjustments in order to integrate it with the NDFS cycle detection algorithm from Section 2.2.

Given a LTS $T = (S, R, L, A, \hat{s}, F)$ let $\text{Perm}(I)$ and $\text{Perm}(S)$ be the groups of permutations of the sets $I = \{1, \ldots, N\}$ of pids and $S$ of states, respectively. Both $\text{Perm}(I)$ and $\text{Perm}(S)$ are groups under the functional composition $\circ$ defined as: For any two permutations $\pi_1, \pi_2$, $(\pi_1 \circ \pi_2)(x) \equiv \pi_1(\pi_2(x))$. We denote with $e$ the identity permutation and $\pi^{-1}$ is the inverse of $\pi$. Further, we assume that each permutation $\pi \in \text{Perm}(I)$ can be lifted into the permutation $\pi^* \in \text{Perm}(S)$ on the state set $S$. This assumption is quite natural regarding the way symmetry reduction is handled in practice (see for instance [19, 10, 3]). Formally, we require that there exists a mapping $(\cdot)^* : \text{Perm}(I) \rightarrow \text{Perm}(S)$ which maps each $\pi \in \text{Perm}(I)$ into $\pi^* \in \text{Perm}(S)$.

**Definition 7.** Given a LTS $T = (S, R, L, A, \hat{s}, F)$ and a mapping $(\cdot)^* : \text{Perm}(I) \rightarrow \text{Perm}(S)$, a subgroup $G$ of $\text{Perm}(I)$ is called a symmetry group of $T$ iff for all $\pi \in G$

- $s \xrightarrow{a} s' \in R$ iff $\pi^*(s) \xrightarrow{a} \pi^*(s') \in R$ and $\text{Pid}(\pi^*(s) \xrightarrow{a} \pi^*(s')) = \pi(\text{Pid}(s \xrightarrow{a} s')).$
- For all $s \in S \pi^*(s) = \hat{s}$ iff $s = \hat{s}$
- For all $s \in S L(s) = L(\pi^*(s))$.
- $s \in F$ iff $\pi^*(s) \in F$.

We say that the states $s_1, s_2 \in S$ are in the same orbit iff there exists $\pi \in G$ such that $\pi^*(s_1) = s_2$. The symmetry group $G$ induces the orbit relation $\Theta_G \subseteq S \times S$ defined as $\Theta_G = \{(s_1, s_2) \mid s_1$ and $s_2$ are in the same orbit$\}$. From the definition of symmetry it is trivial to show that $\Theta_G$ is a bisimulation on $T$. Thus, it follows that each function $h : S \rightarrow S$ which has the following two properties

- $(s, h(s)) \in \Theta$.
- If $(s, s') \in \Theta$, then $h(s) = h(s')$

is a bisimulation preserving selection function on $T$. In what follows we assume that $h$ denotes a bisimulation selection function which satisfies the aforementioned two properties.

---

\footnote{Roughly speaking, each atomic proposition in the property formula (Büchi automaton) should be invariant under $h$, in order for $L(s) = L(h(s))$ to hold.}
4.4 Annotated LTS

As has been discussed above, the abstract LTS obtained according to Def. 5 cannot be used to detect fair acceptance cycles in the original LTS. The obvious reason for that is that in general a transition in \( h(T) \) represents several transitions from \( T \), which has as a consequence that the Pids of the original transitions are lost. For instance, in Fig. 10 transition \( s_2 \xrightarrow{1} s_2 \) of \( h(T) \) represents the transitions \( s_2 \xrightarrow{1} s_3 \) and \( s_3 \xrightarrow{2} s_2 \) of \( T \) given in Fig. 9. In order to solve this problem, we define along the lines of [11] a less compressed version of the reduced LTS from Def. 5. Using some additional information about the transitions it is possible to recover the paths in the original LTS \( T \). More precisely, it is possible by passing along an acceptance cycle in \( h(T) \) to recover a corresponding fair acceptance cycle in \( T \).

To this end first notice that for any two states \( s, s' \in S \) belonging to the same orbit there can exist several permutations \( \pi \in G \) such that \( \pi^*(s) = s' \).

Given a state \( s \) and its (orbit) representative \( h(s) \), from the permutations \( \pi \) such that \( \pi^*(s) = h(s) \) we choose according to some criterion a (unique) canonical permutation \( \pi_s \). For brevity, in the sequel, given a state \( s \) and a permutation \( \pi \), we write \( \pi(s) \) instead of \( \pi^*(s) \). One can say that \( \pi_s \) is the encoding permutation which encodes state \( s \) from \( T \) into its representative \( h(s) \) in \( h_G(T) \), while \( \pi_s^{-1} \) is the decoding permutation which recovers \( s \) when applied to \( h(s) \). In the new version of the abstract LTS we annotate the transitions with the decoding canonical permutation which we use later in order to regenerate paths of the original LTS \( T \).

**Definition 8.** Given a LTS \( T = (S, R, L, A, \hat{s}, F) \) with a symmetry group \( G \) and a selection function \( h : S \to S \) we define the corresponding annotated quotient LTS \( h_G(T) \) to be \( (S_h, R_h, L_h, A_h, h(\hat{s}), F_h) \), where

- \( S_h = h(S) \),
- \( A_h = A \times G \),
- \( r_1 \xrightarrow{a, \pi^{-1}} r_2 \in R_h \subseteq S_h \times A_h \times S_h \) iff \( r_1 \xrightarrow{a} s \in R \) and \( h(s) = r_2 \), with \( Pid(r_1 \xrightarrow{a, \pi^{-1}} r_2) = Pid(r_1 \xrightarrow{a} s) \). (Intuitively, the transition \( r_1 \xrightarrow{a} s \in R \) is represented by \( r_1 \xrightarrow{a, \pi^{-1}} r_2 \in R_h \).)
- for all \( r \in S_h \), \( L_h(r) = L(r) \), and
- \( F_h = h(F) \).

**Example 2.** The annotated quotient LTS for the LTS \( T \) in Fig. 9 is given in Fig. 11. (Recall that we label the transitions with their pids instead of actions.) There are only two processes in the system. The set \( G = \{ e, f \} \), where \( e \) is the

---

5 For instance, consider the following situation. Let processes \( i, j \) and \( k \) be instances of the same program text and let all these processes have access only to local variables. Further, let in some state \( s \) the program counters and the local variables of these processes have the same value. This means that except for the pids, the parts of the processes in the state description are the same. Obviously, the permutation which only swaps processes \( i \) and \( j \) will produce the same state as the permutation which only swaps \( i \) and \( k \).
identity permutation, and \( f \) is the “flip” permutation which maps 1 to 2 and vice versa. (Notice that \( e^{-1} = e \) and \( f^{-1} = f \).)

Using the permutations we are able to “unwind” the paths from \( h_G(T) \) transition by transition into paths in \( T \). Given a path \( p = r_0 \xrightarrow{a_0,p_1} r_1 \xrightarrow{a_1,p_2} \ldots \xrightarrow{a_{n-1},p_n} r_n \) we define the “cumulative” permutation along \( p \) up to the state \( r_i \)

\[
\Pi_{p,i} = \begin{cases} 
\pi_1 \circ \pi_2 \circ \ldots \circ \pi_i, & 1 \leq i \leq n \\
1, & i = 0
\end{cases}
\]

The following lemma from [11] establishes a more precise correspondence between the paths in \( T \) and \( h_G(T) \):

**Lemma 7 ([11]).**

1. If \( p = r_0 \xrightarrow{a_0,p_1} r_1 \xrightarrow{a_1,p_2} \ldots \xrightarrow{a_{n-1},p_n} r_n \) is a path in \( h_G(T) \), then the path obtained by unwinding \( p \),

\[
\text{unwind}(p) \overset{\text{def}}{=} r_0 \xrightarrow{a_0} \Pi_{p,1}(r_1) \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} \Pi_{p,n}(r_n)
\]

where \( \text{Pid}(\Pi_{p,i-1}(r_{i-1}) \xrightarrow{a_{i-1}} \Pi_{p,i}(r_i)) = \Pi_{p,i-1}(\text{Pid}(r_{i-1} \xrightarrow{a_{i-1}} r_i)), 1 \leq i \leq n, \) is a path in \( T \). (Notice that \( r_0 = \Pi_{p,0}(r_0) \).)

2. If \( p = r_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n \) is a path in \( T \) starting at a representative state \( r_0 \), then there exists a path in \( h_G(T) \)

\[
q = r_0 \xrightarrow{a_0,p_1} r_1 \xrightarrow{a_1,p_2} \ldots \xrightarrow{a_{n-1},p_n} r_n
\]

such that \( p = \text{unwind}(q) \).

Intuitively, an edge permutation \( \pi_i \) can be seen as a “relative” decoding which applied to the representative state \( r_i \) recovers the original state \( s_i \), under the assumption that the processes are not permuted in the state \( r_{i-1} \). In contrast, \( \Pi_{p,i} \)
is an “absolute” decoding which restores $s_i$ taking into account the cumulative effect of all permutations from the start state $r_0$.

Having the permutation on the edges, one can recover fair acceptance cycles in $T$:

Example 3. The acceptance cycle $s_1 \overset{1.c}{\rightarrow} s_2 \overset{1.f}{\rightarrow} s_2 \overset{1.c}{\rightarrow} s_1$, which is not fair in $h_G(T)$ (Fig. 11), is unwound into the fair acceptance cycle $s_1 \overset{1}{\rightarrow} s_2 \overset{1}{\rightarrow} s_3 \overset{1}{\rightarrow} s_2 \overset{1}{\rightarrow} s_1$ in $T$ (Fig. 9).

Given a cycle $c$ and some integer $n \geq 1$, then $c^n$ denotes the cycle obtained by concatenating $n$ copies of $c$. Given a path $p$ and a cycle $c$ we say that $p$ is along $c$ iff there exists an integer $n \geq 1$, and paths (possibly with length 0) $q, q'$ such that $c^n = qPq'$. If $p$ is a cycle, then we say that $p$ is a subcycle of $c$.

In order to establish a correspondence with the fair cycles in $T$, we need a new kind of fair cycles in $h_G(T)$ called subtly fair cycles [11]. We would like to consider as fair those cycles in $h_G(T)$ that can be unwound into fair cycles in $T$ as implied by the following definition:

Definition 9. Cycle $c$ in $h_G(T)$ is subtly (weakly) fair iff for each state $r$ in $c$ and for each $i \in I$ there exists a path $p = r_0 \overset{a_0, \pi_1}{\rightarrow} r_1 \cdots \overset{a_{n-1}, \pi_n}{\rightarrow} r_n \overset{a_n, \pi_{n+1}}{\rightarrow} r_{n+1}$ along $c$, such that

- $r = r_0$,
- process $j$, where $j = \Pi_{p,n}^{-1}(i)$, is disabled or executed in $r_n$ (i.e., $\text{Pid}(r_n \overset{a_{n+1}, \pi_{n+1}}{\rightarrow} r_{n+1} = j)$).

In order to grasp the intuition behind the above definition first notice that, because the pids are scrambled in $h_G(T)$, the process with pid $i$ in $r_0$ appears “disguised” in each state $r_k$ ($0 \leq k \leq n + 1$) of $p$ as process with pid $l$, where $l = \Pi_{p,k}^{-1}(i)$. In other words, process $l$ is the encoding in $r_k$ of process $i$. Thus, if we take the pids in $r_0$ as a reference point, when we detect that process $j = \Pi_{p,n}^{-1}(i)$ is executed or disabled in $r_n$, this actually means that process $i$ (which we are tracing along $p$) is the one which is executed or disabled.

Another useful observation is that $j = \Pi_{p,n}^{-1}(i)$ is the same as $\Pi_{p,n}(j) = i$, i.e., the decoding of pid $j$ is $i$. By Lemma 7, the path $p$ in $h_G(T)$ from the definition above mimics a path $p'$ in the original LTS $T$, such that both $p$ and $p'$ begin in the same state $r_0$. Thus, it is really process $i$ which is executed or disabled in $p'$, but in the annotated LTS the same process occurs encoded as process $j$ (which is executed or disabled in the corresponding abstract path $p$).

Note 1. The requirement from Def. 9 is satisfied for each state in $c$ iff it is satisfied for some state in $c$. Suppose that the requirement is satisfied for some state $r$, and let $p' = r_0 \overset{a_0, \pi'_1}{\rightarrow} r'_1 \cdots \overset{a_{k-1}, \pi'_k}{\rightarrow} r'_k$ be a path along $c$ from some arbitrary state $r'$ to $r = r'_k$. Let $m = \Pi_{p',k}^{-1}(i)$, for a given $i \in I$. As $r$ satisfies the requirement of the definition, there is some path $p = r_0 \overset{a_0, \pi_1}{\rightarrow} r_1 \cdots \overset{a_{n-1}, \pi_n}{\rightarrow} r_n \overset{a_n, \pi_{n+1}}{\rightarrow} r_{n+1}$ along $c$, where $r_0 = r$, such that $j = \Pi_{p,n}^{-1}(m)$.
is disabled or executed in \( r_n \). Obviously the path obtained by concatenating \( p' \) and \( p \) satisfies the requirements of the definition for process \( i \) with regard to \( r' \) because process \( j = \Pi_{p,n}^{-1}(\Pi_{p',k}(i)) = (\Pi_{p',k} \circ \Pi_{p,n})^{-1}(i) = \Pi_{p,p,k+n}^{-1}(i) \) is disabled or executed in \( r_n \).

The next two Lemmata follow directly from Def. 9 and Lemma 7:

**Lemma 8.** If a given cycle \( c = r_0 s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} \ldots s_{n-1} \overset{a_{n-1}}{\rightarrow} r_0 \) in \( T \) is fair and \( r_0 = h(r_0) \), i.e., \( r_0 \) is a representative state, then there exists in \( h_G(T) \) a subtly fair cycle \( c' \), such that \( c = \text{unwind}(c') \).

**Proof.** Let \( c = r_0 s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} \ldots s_{n-1} \overset{a_{n-1}}{\rightarrow} r_0 \) be a fair cycle in \( T \) beginning at the representative state \( r_0 \). By point 2 of Lemma 7 there exist a corresponding cycle in \( h_G(T) \)

\[ c' = r_0 s_0 \overset{a_0,\pi_1}{\rightarrow} r_1 s_1 \overset{a_1,\pi_2}{\rightarrow} \ldots r_{n-1} s_{n-1} \overset{a_{n-1},\pi_n}{\rightarrow} r_0 \]

such that \( c = \text{unwind}(c') \). Because \( c \) is fair, for any \( i \in I \) there exist some \( 0 \leq k < n \) such that process \( i \) is disabled or executed in the state \( s_k \) in \( c \). If process \( i \) is executed, then the definition of \( \text{unwind} \) (Lemma 7, item 1) implies that process \( j = \Pi_{c,k}^{-1}(i) \) is executed in \( r_k = \Pi_{c,k}^{-1}(s_k) \). If process \( i \) is disabled in \( s_k \), then from the definition of symmetry (Def. 7) it follows that also process \( j = \Pi_{p,k}^{-1}(i) \) is disabled in \( r_k \). Thus, we can conclude that for the state \( r_0 \) in \( c' \) for each \( i \in I \) there exists a path \( p \) along \( c' \) which satisfies the requirements of the definition of subtly fair cycle (Def. 9). By Note 1 it follows that the requirements of the Def. 9 are also satisfied for all states of \( c' \). \( \square \)

**Lemma 9.** If cycle \( c \) in \( h_G(T) \) is subtly fair, then there exists \( l \geq 1 \) such that the cycle \( \text{unwind}(c') \) is fair in \( T \).

**Proof.** Consider an arbitrary process \( i \in I \). It is clear that \( l \) can be chosen such that the path \( p \) from Def. 9 is a prefix of \( c' \). From Def. 9 we have that process \( j = \Pi_{p,n}^{-1}(i) \) is disabled or executed in the state \( r_n \) in \( p \). If \( j \) is executed in \( r_n \), then the definition of \( \text{unwind} \) implies that \( i = \Pi_{p,n}(j) \) is executed in the state \( s_n = \Pi_{p,n}(r_n) \) from \( \text{unwind}(c') \). If \( j \) is disabled, then by the definition of symmetry, \( i = \Pi_{p,n}(j) \) is disabled in \( s_n \). \( \square \)

Lemma 9 and 8 imply the following theorem:

**Theorem 5.** There exists in \( T \) a reachable fair acceptance cycle iff there exists in \( h_G(T) \) a reachable subtly fair acceptance cycle.

**Proof.** Let \( c = s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} \ldots s_{n-1} \overset{a_{n-1}}{\rightarrow} s_0 \) be a reachable fair acceptance cycle in \( T \), such that \( s_0 \) is not necessarily a representative state. Consider the cycle obtained by applying the canonical permutation \( \pi_{s_0} \) to each transition of \( c: \pi_{s_0}(c) = \pi_{s_0}(s_0) \overset{a_0}{\rightarrow} \pi_{s_0}(s_1) \overset{a_1}{\rightarrow} \ldots \pi_{s_0}(s_{n-1}) \overset{a_{n-1}}{\rightarrow} \pi_{s_0}(s_0) \), with \( Pid(\pi_{s_0}(s_k) \overset{a_k}{\rightarrow} \pi_{s_0}(s_{k+1})) = \pi_{s_0}(Pid(s_k \overset{a_k}{\rightarrow} s_{k+1}))(1 \leq k < n) \). The cycle \( \pi_{s_0}(c) \) is in \( T \) by the definition of symmetry (Def. 7). By the fairness of \( c \) for each \( i \in I \) the process \( \pi_{s_0}(i) \) is enabled or executed in \( \pi_{s_0}(c) \), which implies that \( \pi_{s_0}(c) \) is also fair. As
Recall that because the pids are scrambled in \( h \) between the fair cycles. However, extracting the fair acceptance cycles directly by the definition of symmetry (Def. 7), \( \hat{s} = h(\hat{s}) \). Thus, the second part of Lemma 7 implies that there exists a corresponding path in \( h_G(T) \) from the initial state \( h(\hat{s}) \) to \( h(s_0) \), i.e., that \( c' \) is reachable.

For the reverse direction, let \( c \) be an acceptance cycle in \( h_G(T) \). The existence of the corresponding cycle \( c' \) in \( T \) is given by Lemma 9. From the definition of the annotated LTS it easy to see that \( c' \) must contain acceptance states. In order to prove the reachability of \( c' \) in \( T \) consider the representative state \( r \) in which we begin the unwinding of \( c \) according to Lemma 9. As \( c \) is reachable there exists a path \( p \) in \( h_G(T) \) from \( h(\hat{s}) \) to \( r \). Because \( h(\hat{s}) = \hat{s} \), by the first part of Lemma 7 there exists the path \( unwind(p) \) in \( T \) from \( \hat{s} \) to some state \( s \) such that \( r = \pi(s) \) for some \( \pi \in G \). By the definition of symmetry, \( \pi_s(p) \), defined analogously as \( \pi_s(c) \) above, is a path in \( T \) from \( \pi_s(\hat{s}) = \hat{s} \) to \( \pi_s(s) = r \). Thus, \( r \), and therefore \( c' \), is reachable in \( T \).

\[ \square \]

4.5 Threaded LTS

Thanks to the preserved permutations we are able to establish a correspondence between the fair cycles. However, extracting the fair acceptance cycles directly from \( h_G(T) \) can still be difficult. This is mainly because during the state space search in \( h_G(T) \) we might have to pass several times through the same (representative) state. In order to get the intuition behind the problem consider the path \( p \) from Def. 9 which we construct for detecting some subtly fair cycle \( c \).

Recall that because the pids are scrambled in \( h_G(T) \), process \( i \) (from the state \( r_0 \) which we are tracing along the path \( p \), in the state \( r_k \) \((0 \leq k \leq n + 1)\) appears "disguised" as process \( l = \Pi^{-1}_{p, k}(i) \). As in general \( \Pi^{-1}_{k_1, p}(i) \neq \Pi^{-1}_{k_2, p}(i) \) \((0 \leq k_1 < k_2 \leq n + 1)\), it can happen that two states \( r_{k_1} \) and \( r_{k_2} \) of \( p \) are the same, but process \( i \) has different encodings in each of them. Obviously, the different occurrences in \( p \) of the same state \( r \) from the cycle \( c \) are distinguished by the pid \( l \) of the process which is the encoding in \( r \) of process \( i \). In order to overcome this problem we define another, less compressed, version of \( h_G(T) \).

The new kind of LTS is a straightforward adaptation of the notion of threaded graph from [11]. The new threaded LTS physically implements the annotating permutations. To this end, the states are extended with a component which takes values from the set \( I \). In this way one can distinguish between the above mentioned different appearances of the same state along some path. In the new LTS there exists a transition between two states \((r_1, i), (r_2, j)\) if and there is a transition between \( r_1 \) and \( r_2 \) in \( h_G(T) \) annotated with \( \pi \) and \( j = \pi^{-1}(i) \). As \( \pi^{-1} \) is the "relative" encoding permutation, this makes it easier to keep track of a particular process \( i \) in the reduced state space. Because the original states from \( T \) are scrambled in \( h_G(T) \), process \( i \) in \( r_1 \) appears in \( r_2 \) encoded as process \( j \).

The second components of the states in \( h^*_G(T) \) reflect this transformation.
Definition 10. The threaded LTS associated with $h_G(T)$ is the LTS $h^*_G(T) = (S^*_h, R^*_h, L^*_h, A^*_h, \hat{s}^*, F^*_h)$, where

- $S^*_h = (S_h \times I) \cup \{\hat{s}^*\}$,
- $A^*_h = A \cup \{\tau\}$ (Note that, unlike $A_h$, the actions of $A^*_h$ do not contain permutations from $G$ as a second component),
- for all $i \in I$, $\hat{s}^* \xrightarrow{\tau} (h(\hat{s}), i) \in R^*_h \subseteq S^*_h \times A^*_h \times S^*_h$ and $\text{Pid}(\hat{s}^* \xrightarrow{\tau} (h(\hat{s}), i)) = 0$,
- $(r_1, i) \xrightarrow{a} (r_2, j) \in R^*_h \subseteq S^*_h \times A^*_h \times S^*_h$ iff $r_1 \xrightarrow{a,\pi} r_2 \in R_h$, $j = \pi^{-1}(i)$, and $\text{Pid}((r_1, i) \xrightarrow{a} (r_2, j)) = \text{Pid}(r_1 \xrightarrow{a,\pi} r_2)$ (Note that the transition pids are preserved),
- for all $(r, i) \in S^*_h, L^*_h((r, i)) = L_h(r)$, and $L^*_h(\hat{s}^*) = L_h(h(\hat{s}))$.

Note 2. The initial state $\hat{s}^*$ is only used to represent that the search in $h^*_G(T)$ can start nondeterministically in any copy $(h(\hat{s}), i)$ of the initial state of $h_G(T)$, and it does not play any role in the sequel. There are only transitions from but not to the initial state $\hat{s}^*$. Also the Pids of the transitions from the initial state could have been assigned arbitrarily.

Example 4. The threaded graph $h^*_G(T)$ corresponding to $h_G(T)$ from Fig. 11 is given in Fig. 12. Notice that $h^*_G(T)$ is not symmetric in its pid labeling as

![Threaded graph example](image_url)

the original LTS $T$ is. This is because the pids of the transitions from $h_G(T)$ are preserved in $h^*_G(T)$. In other words, the pids of the image transitions in $h^*_G(T)$ are not affected by the annotating $h_G(T)$. On the other hand, the second components of the states in $h^*_G(T)$ do depend on the annotating permutations,
which causes the asymmetry. (Although in this particular case, for \( N = 2 \), the threaded LTS has greater size than the original LTS \( T \), in practice, for greater values of \( N \), the LTS \( h_G^* (T) \) is much smaller than \( T \). We revisit this issue later when we discuss the complexity of the algorithm which combines symmetry with fairness.)

Given a process \( i \) and a path \( p \) in \( h_G (T) \), we define in \( h_G^* (T) \) a corresponding “thread” \( q \), which is actually the path \( p \) whose states are extended such that their second components reflect the encodings of process \( i \). More precisely, if \( p \) begins in the state \( r_0 \), then \( q \) begins in \((r_0, i)\). In each state \((r_k, i_k)\) of \( q \) the second component \( i_k \) is the pid of the process which is the encoding in \( r_k \) of the process \( i \) from \( r_0 \).

The following lemma, analogous to Lemma 3.7 from [11], follows directly from Def. 10. It establishes the correspondence between the paths in \( h_G (T) \) and their threads in \( h_G^* (T) \):

**Lemma 10.** 1. If \( p = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_n \) is a path in \( h_G (T) \) and \( i \in I \), then the path

\[
\text{thread}(p, i) \overset{\text{def}}{=} (r_0, i) \xrightarrow{a_0}(r_1, \Pi_{p,1}(i)) \xrightarrow{a_1}(r_2, \Pi_{p,2}(i)) \ldots (r_{n-1}, \Pi_{p,n-1}(i)) \xrightarrow{a_{n-1}}(r_n, \Pi_{p,n}(i))
\]

with \( \text{Pid}((r_k, \Pi_{p,k}(i)) \xrightarrow{a_k}(r_{k+1}, \Pi_{p,k+1}(i))) = \text{Pid}(r_{k+1} \xrightarrow{a_{k+1}} r_{k+1}), 0 \leq k < n \), is the corresponding thread of \( p \) in \( h_G^* (T) \) with respect to process \( i \).

2. If \( p = (r_0, i_0) \xrightarrow{a_0}(r_1, i_1) \xrightarrow{a_1} \ldots (r_{n-1}, i_{n-1}) \xrightarrow{a_{n-1}}(r_n, i_n) \) is a path in \( h_G^* (T) \), then there exists a corresponding path in \( h_G (T) \)

\[
q = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_n
\]

such that \( p = \text{thread}(q, i_0) \).

**Proof.** 1. Let \( p \) and \( i \) be as defined in the lemma. We prove the existence of \( \text{thread}(p, i) \) by induction on the length of \( p \). The base case, when \( p \) consists of only one transition, i.e., \( n = 1 \), follows directly by Def. 10, by observing that \( \Pi_{p,1}(i) = \pi_1^{-1} \). Suppose that there exists a thread with respect to \( i \) for each \( n = k \) and let \( p = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_k \xrightarrow{a_k,\pi_{k+1}} r_{k+1} \) be a path in \( h_G (T) \). By the induction hypothesis to the path \( p' = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{k-1} \xrightarrow{a_{k-1},\pi_{k}} r_k \) there corresponds the thread \( \text{thread}(p', i) \) in \( h_G^* (T) \) which terminates in the state \((r_k, \Pi_{p,k}^{-1}(i))\). By Def. 10 to the transition \( r_k \xrightarrow{a_{k},\pi_{k+1}} r_{k+1} \) in \( h_G (T) \) there corresponds in \( h_G^* (T) \) the transition \((r_k, \Pi_{p,k}^{-1}(i)) \xrightarrow{a_k}(r_{k+1}, \pi_{k+1}^{-1}((\Pi_{p,k}^{-1}(i)))\). It is easy to check that \( \pi_{k+1}^{-1}(\Pi_{p,k}^{-1}(i)) = \Pi_{p,k+1}^{-1}(i) \), which implies that indeed \( \text{thread}(p', i) \) can be extended into \( \text{thread}(p, i) \).

2. Using very similar arguments as for point 1 above, we show by induction on the length of \( p \) that for each state \((r_k, i_k)\) \((0 \leq k \leq n)\) in \( p \) there exists a path \( q \) in \( h_G (T) \) such that \( i_k = \Pi_{q,k}^{-1}(i_0) \). For \( n = 1 \) Def. 10 implies immediately that to \((r_0, i_0) \xrightarrow{a_0}(r_1, i_1)\) there corresponds the transition \( r_0 \xrightarrow{a_0,\pi_1} r_1 \) in \( h_G (T) \).
such that $i_z = \pi_1^{-1}(i_0) = \Pi_{q,z}^{-1}(i_0)$. Assume that the claim holds for $n = k$. The induction hypothesis implies that there exists a path $q'$ in $h_G(T)$ that terminates in the state $r_k$. By Def. 10 to the transition $(r_k, i_k) \xrightarrow{a_k} (r_{k+1}, i_{k+1})$ there corresponds the transition $r_k \xrightarrow{a_k, \pi_{k+1}} r_{k+1}$ in $h_G(T)$ such that $i_{k+1} = \pi_{k+1}^{-1}(i_k)$. Thus, with this last transition $q'$ can be extended into the path $q$. The induction hypothesis implies $i_k = \Pi_{q,k}^{-1}(i_0)$. Thus, taking into account that $\pi_{k+1}^{-1}(\Pi_{q,k}^{-1}(i_0)) = \Pi_{q,k+1}^{-1}(i_0)$, it is obvious that $q$ is the wanted path. 

Intuitively, given a path $p$ in $h_G(T)$ and a pid $i$, along thread($p, i$) one can follow the evolution of $i$ under the cumulative permutation starting at the initial state $r_0$ of the path $p$. Notice that $\Pi_{p,j}^{-1}(i)$ is exactly the encoding of the process $i$ in the state $r_j$ on $p$.

The next step is to establish a correspondence between the subtly fair acceptance cycles from $h_G(T)$ and the acceptance cycles in $h^*_G(T)$. We first describe informally the basic idea. Each subtly fair cycle in $h_G(T)$ contains $N$ subcycles $c_i$ ($i \in I$) such that (1) $c_i$ corresponds to process $i$ and (2) $c_i$ is subtly fair with respect to process $i$ in the sense that there exists a state $r$ on $c_i$ such that the process which is encoding of process $i$ in $r$ is executed or disabled in $r$. Now, showing that the cycle $c$ is subtly fair boils down to showing for each process $i \in I$ that its corresponding subcycle $c_i$ is subtly fair with respect to it. In order to be sure that in our check we cover all the processes from $I$, i.e., that no two subcycles correspond to the same process, we require that the subcycles share a common (acceptance) state $r'$. (One can consider this state as a reference point from which we start the paths $p$ from the definition Def. 9.) As each subcycle $c_i$ has a corresponding image (thread) $c^*_i$ in $h^*_G(T)$, we check the subtle fairness of $c_i$ by checking that $c^*_i$ is fair in the sense which is defined later.

Thus, from the discussion above it follows that we need new kinds of fairness. We begin with the following definition of a cycle which is fair with respect to one particular process:

**Definition 11.** Cycle $c$ in $h_G(T)$ is subtly (weakly) fair with respect to process $i$, $i \in I$, iff there exists a state $r$ of $c$ and a path along $c$ starting at $r = r_0$, $p = r_0 \xrightarrow{a_0, \pi_1} r_1 \cdots \xrightarrow{a_n, \pi_n} r_n \xrightarrow{a_n, \pi_{n+1}} r_{n+1}$, such that process $j = \Pi_{p,n}^{-1}(i)$ is disabled in $r_n$ or it is executed in $c$ in $r_n$, i.e., Pid($r_n \xrightarrow{a_n, \pi_{n+1}} r_{n+1}$) = $j$.

The following obvious result is essentially an alternative definition of a fair acceptance cycle in $h_G(T)$:

**Lemma 11.** Cycle $c$ in $h_G(T)$ is subtly fair iff for each process $i$, $i \in I$ there exists a subcycle of $c$, $c_i$, such that $c_i$ is subtly fair with respect to process $i$.

**Proof.** Let $c$ be a subtly fair cycle and let $r$ be an arbitrary state in $r$. By Def. 9 there exists a path $p$ along $c$, which begins in $r$, as defined in Def. 9. Consider an arbitrary process $i$. From the definition of subtly fair cycle with respect to a given process (Def. 11) it is clear that one can use the same state $r$ and path $p$
in order to satisfy Def. 11 for \( c \). Thus, \( c \) is also subtly fair with respect to process \( i \). Since \( c \) is subcycle of itself by taking \( c_i = c \) we obtain the the desired subcycle for each \( i \in I \).

For the reverse direction, given a cycle \( c \) in \( h_G(T) \), let us assume that there exist subcycles \( c_i \ (i \in I) \) of \( c \) such that \( c_i \) is subtly fair with respect to process \( i \). Consider the cycle \( c_i \) for an arbitrary \( i \). By the definition of a subtly fair cycle with respect to a single process (Def. 11) there exists a state \( r \) of \( c \) and a path \( p \) along \( c \) which have the properties required in that definition. As \( c_i \) is a part of \( c \) the path \( p \) is also along \( c \). Taking into account Note 1, if such a path \( p \) exists from the state \( r \) on \( c \), then there is a path with the same properties for all states in \( c \). Therefore, Def. 9 is satisfied for process \( i \). As \( i \) was arbitrary chosen, the same argument can be repeated for all \( i \in I \), which implies that \( c \) is a subtly fair cycle.

\[ \square \]

Note 3. Obviously cycle \( c \) in \( h_G(T) \) is subtly fair iff it is subtly fair with respect to each process \( i \in I \). Also, given a state \( s \) on \( c \) one can always choose the subcycles \( c_i \) such that \( s \) belongs to all of them.

In a similar way like for \( h_G(T) \), we need to adapt the definition of fairness to \( h^*_G(T) \):

**Definition 12.** Cycle \( c \) in \( h^*_G(T) \) is plainly fair with respect to process \( i \) iff there exist in \( c \) a state \( (r, i) \) and a state \( (r', j) \) in which process \( j \) is disabled or it is executed in \( c \).

The intuition is that the cycle \( c \) corresponds to the cycle \( c' \) in \( h_G(T) \) obtained by omitting the second components of the states in \( c \). In terms of Def. 9, we can construct the path \( p \) along the cycle \( c \) (i.e., \( c' \)), with \( r_0 = r \) (i.e, \( (r, i) \)) and \( r_n = r' \), (i.e., \( (r', j) \)). Obviously, \( j = \Pi_{p,n}^{-1}(i) \), as required by Def. 9.

It is easy to derive from Lemma 10 the following correspondence between cycles of \( h_G(T) \) and \( h^*_G(T) \) which are fair with respect to a particular process:

**Lemma 12.** For all \( i \in I \), cycle \( c \) in \( h_G(T) \) is a subtly fair with respect to process \( i \) iff there exists in \( h^*_G(T) \) a cycle which is plainly fair with respect to process \( i \).

*Proof.* Let \( c = r_0 \xrightarrow{a_0,\pi_1} r_1 \xrightarrow{a_1,\pi_2} \ldots r_{n-1} \xrightarrow{a_{n-1},\pi_n} r_0 \) be a subtly fair with respect to process \( i \) cycle in \( h_G(T) \). The path in \( h^*_G(T) \)

\[ \text{thread}(c, i) = (r_0, i) \xrightarrow{a_0} (r_1, \Pi_{p,1}^{-1}(i)) \xrightarrow{a_1} \ldots (r_{n-1}, \Pi_{p,n-1}^{-1}(i)) \xrightarrow{a_{n-1}} (r_n, \Pi_{p,n}^{-1}(i)) \]

need not be a cycle because in general \( \Pi_{p,n}^{-1}(i) \neq i \). However, for every permutation \( \pi \in G \) there exists \( l \geq 1 \) such that \( \pi_l = e \). Consequently, for some \( l \) it holds \( (\Pi_{p,l}^{-1})'(i) = i \). This implies that \( \text{thread}(c', i) \) (which begins at \( (r_0, i) \)) is a cycle in \( h^*_G(T) \). Because \( c \) is subtly fair with respect to process \( i \), there exists some \( k \) such that process \( j = \Pi_{c',k}^{-1}(i) \) will be executed or disabled in some state \( r_k \) in \( c' \). If process \( j \) is executed in \( r_k \), then by item 1 of the path correspondence
Lemma 10 it is clear that also process $j$ is executed in the state $(r_k, j)$ of the thread of $c'$. By the definition of the threaded LTS (Def. 10) process $j$ is disabled in $r_k$ iff it is disabled in $(r_k, j)$. Thus, the thread of $c'$ is plainly fair with respect to process $i$.

Conversely, let us assume that $c = (r_0, i_0) \xrightarrow{a_0} (r_1, i_1) \xrightarrow{a_1} \cdots (r_{n-1}, i_{n-1}) \xrightarrow{a_{n-1}} (r_0, i_0)$, for an arbitrary $i = i_0$, is a plainly fair cycle in $h_G^r(T)$. By Lemma 10 there exist a cycle $c'$ such that $c = \text{thread}(c', i)$. Because $c$ is plainly fair with respect to $i$ there exists $k$ such that process $i_k$ is executed or disabled in $(r_k, i_k)$. By repeating the reasoning from the proof of the path correspondence Lemma 10 one can show that $i_k = \Pi_{i', k}^{-1}(i_0)$, and consequently $i_k$ will play the role of $j$ from the definition of subtly fair cycle (Def. 9). By the definition of thread it follows that if process $i_k$ is executed in $(r_k, i_k)$, then $i_k$ is executed in the state $r_k$ of $c'$. By the definition of threaded LTS we have that process $i_k$ is disabled in $r_k$ iff it is disabled in $(r_k, i_k)$. Thus, $c'$ is subtly fair with respect to process $i$.

Example 5. Consider the cycle $c = s_1 - s_2 - s_2 - s_2 - s_1$ of $h_G(T)$ in Fig. 11. Its subcycle $c_1 = s_1 - s_2 - s_1$ is fair with respect to process 1, while $c$ itself is subtly fair with respect to process 2. In the threaded LTS $h_G^r(T)$ (Fig. 12) the cycle $(s_1, 1) - (s_2, 1) - (s_1, 1)$, which is plainly fair with respect to process 1, and the cycle $(s_1, 2) - (s_2, 1) - (s_2, 2) - (s_1, 2)$, which is plainly fair with respect to process 2, correspond to $c_1$ and $c$, respectively.

Finally, combining the previous results one can establish the following theorem which is the basis of our algorithm. It states the cycle correspondence between the original LTS $T$ and the corresponding threaded LTS $h_G^r(T)$.

**Theorem 6.** There exists a reachable fair acceptance cycle in $T$ iff for some reachable acceptance state $s$ in $T$ there exists in $h_G^r(T)$ for each $i \in I$ an acceptance cycle $c_i$ which is plainly fair with respect to process $i$ and contains the (acceptance) state $(h(s), i)$.

**Proof.** Assume that there exist a reachable fair acceptance cycle in $T$. By Theorem 5 there exists a reachable subtly fair acceptance cycle $c$ in $h_G(T)$. By Lemma 11 this is equivalent with the existence in $h_G(T)$ of $N$ subcycles of this acceptance cycle, each corresponding to a particular process. (Note that in this direction we did not need explicitly the acceptance state $s$.) By Lemma 12 to each subcycle $c_i$, which is subtly fair with respect to process $i$ ($i \in I$), there corresponds a thread in $h_G^r(T)$ which is plainly fair with respect to $i$.

For the reverse direction assume that there exists in $h_G^r(T)$ a set of $N$ cycles $c_i$, such that $c_i$ is plainly fair with respect to process $i$ ($i \in I$) and each of them contains the acceptance state $(h(s), i)$. According to Lemma 12 for each $c_i$ there exists a subtly fair cycle $c'_i$ in $h_G(T)$. From Lemma 10 it follows that $h(s)$ is in $c'_i$. Now Lemma 11 implies that $c'_i$ can be combined into one subtly fair acceptance cycle in $h_G(T)$. As $s$ is reachable in $T$, it follows by Lemma 7 (point 2) that $h(s)$ is also reachable in $h_G(T)$, i.e., $c'$ is reachable in $h_G(T)$. Finally, Theorem 5 implies the existence of a reachable fair acceptance cycle in $T$. \[\square\]
4.6 An Algorithm for Symmetry Reduction with Weak Fairness

From the previous theorem it is clear that we need an algorithm for finding for each process $i \in I$ an acceptance cycle in $h^*_G(T)$ which is plainly fair with respect to it. This can be done by modifying the cycle check in the NDFS algorithm. Thus, starting in each acceptance state $s$, instead of checking for one (plainly fair) acceptance cycle, we need to check for $N$ acceptance cycles passing through $s$.

Finding a Cycle which is Plainly Fair with with Respect to one Process. We first present an algorithm which can be used for checking for a cycle which is plainly fair with respect to one particular process. Such a cycle check boils down to a relatively simple graph algorithm, given in Fig. 13, which is a variant of (preorder) nested depth first search. Started at the root node (in our

/* seed == r, where r is the root node */
proc dfs2(s)
  add s to Stack1
  add {s,0} to Nodes
  for each transition (s,a,s') do
    add {{s,0},a,{s',0}} to Edges
    if special(s') then
      if {s',1} not in Nodes then dfs3(s') fi
    else
      if {s',0} not in Nodes then dfs2(s') fi
    fi
  od
  delete s from Stack1
end

proc dfs3(s)
  add s to Stack2
  add {s,1} to Nodes
  for each transition (s,a,s') do
    add {{s,1},a,{s',1}} to Edges
    if s' == seed then report cycle
    else if {s',1} not in Nodes then dfs3(s',1) fi
  od
  delete s from Stack2
end

Fig. 13. Finding a cycle passing through the initial state and a special node.

case: an acceptance state) $r$, the algorithm finds a cycle which passes through $r$ and which contains at least one special node (in our case: a state in which process $i$ is disabled or a state which is generated via a transition by process $i$).
The procedures \texttt{dfs2} and \texttt{dfs3} work in separate state spaces. The cycle check is started by calling \texttt{dfs2} at the root node \( r \). If a special node \( s \) is encountered, then \texttt{dfs3} is called. The search continues in the second state space until (the copy of) \( r \) is matched (which means that the desired cycle is detected), or all states reachable from \( s \) are generated, without detecting a cycle, in which case the control is returned back to \texttt{dfs2}. Unlike in the standard NDFS algorithm, in the algorithm in Fig. 13 \texttt{seed} is a constant equal to \( r \). The following theorem states formally the correctness of the algorithm:

**Lemma 13.** Given a graph \( G \), a node \( r \) of \( G \), and a set of special nodes \( S \) (which is a subset of the nodes of \( G \)), the algorithm in Fig. 13, when called on \( r \), reports a cycle iff there exists a cycle in \( G \) which contains \( r \) and at least one special node \( s \in S \).

**Proof.** It is clear that if the algorithm reports a cycle, then it has found a path back to the root \( r \). As the cycle is reported only by \texttt{dfs3}, and as \texttt{dfs3} is called only when a special node is generated, it follows that the detected cycle contains at least one node \( s \in S \).

For the opposite direction, let us suppose that there exists a cycle which contains \( r \) and some state \( s \in S \). Let \( Q \subseteq S \) be the set of special nodes which are in some cycle passing through \( r \). First notice that for each node which is generated in \texttt{dfs2} all its successors are explored either by calling \texttt{dfs2} or \texttt{dfs3} (lines 6-10). This and the correctness of the DFS algorithm, guarantees that, unless the algorithm is stopped by reporting a cycle, each state reachable from \( r \) will be generated at least once (by \texttt{dfs2} or \texttt{dfs3}, or by both). Therefore, also all states in \( S \) will be generated. Moreover, at least one node from \( Q \) will be generated by \texttt{dfs2}. Suppose that this is not true, i.e., suppose that the nodes in \( Q \) are only generated by \texttt{dfs3}. For each such a node \( s \in Q \) there exists a path back to \( r \). On the other hand, \texttt{dfs3} is called on some special node \( s' \in S \) generated by \texttt{dfs2}. As there is a path from \( r \) to \( s' \) and from \( s' \) to \( s \), obviously \( s' \) also must be in \( Q \), which is a contradiction.

Now, let \( s \) be the node in \( Q \) which is the first (in preorder) generated by \texttt{dfs2}. We show that a cycle through \( s \) will be reported. Suppose, this is not true, i.e., that \texttt{dfs3} is wrongly truncated on some node \( s' \) from which there exists a path back to \( r \). This is only possible if \( s' \) has been generated by some previous cycle check from another state \( s'' \in S \). This implies that there exists a cycle passing through \( s'', s', \) and \( r \), i.e., \( s'' \in Q \). As \( s'' \) must have been generated by \texttt{dfs2} before \( s \), this is a contradiction. \( \square \)

**A NDFS Algorithm for Symmetry Reduction with Weak Fairness.**

Now that we have the cycle check algorithm we are ready to proceed with the algorithm for symmetry reduction under weak fairness (RWF), given in Fig. 14. In analogy with the weak fairness (WF) algorithm, we do the search for the plainly fair cycles sequentially, starting from process 1. If a cycle for process \( i \) is found, then we continue by starting the search for a cycle corresponding to
process $i + 1$. In accord with Theorem 6, the algorithm reports a fair acceptance cycle in $T$ if it finds for each $i$ a corresponding cycle $c_i$.

The algorithm is a version of the standard NDFS algorithm – a kind of double nested depth first search. The first DFS is performed by calling $dfs1$ in the initial state of $h_G(T)$. The cycle check is implemented by a straightforward adaptation of the algorithm from Fig. 13. The procedures $dfs2$ and $dfs3$ in Fig. 13 correspond to $dfs2$ and $dfs3$, respectively in RWF. Nodes and Edges are renamed into States and Transitions, respectively. Like in the weak fairness algorithm, as the process identity plays a crucial role, the for statement from line 4 of the algorithm in Fig. 6 is expanded into two nested for iterations (lines 14 and 20 in the RWF algorithm). The function special from line 6 in Fig. 6 is refined into a check if the process is disabled or executed (lines 17 and 21, respectively). Function \( \text{perm}(s', C1) \) returns the value $\pi_{s'}(C1) = (\pi^{-1})_{s'}^{-1}(C1)$ (in accord with Def. 10). The cycle check is started in postorder for each acceptance state by calling $dfs2$.

The cycle check for a particular process $i$ is launched by calling $dfs2$ in the acceptance state $(s, i)$. Procedure $dfs2$ works in a copy of $h_G^*(T)$ corresponding to process $i$. Analogously to the WF algorithm, each state is extended with a counter that denotes the process to which the copy corresponds. If a state is generated in which process $i$ is disabled or executed (lines 17, 18 and 21, 22), then $dfs3$ is called. The latter operates in a copy of $h_G(T)$ (instead of $h_G^*(T)$) corresponding to process $i$, which is also denoted by a counter added to the state. This small optimization is trivially justified by the following result, which is proved along the same lines as Lemma 1 from [14]:

**Lemma 14.** Let there exist a path from $(s, j)$ to $(s', j')$ in $h_G^*(T)$. Then there exists a path in $h_G^*(T)$ from $(s', j')$ to $(s, j)$ iff there exists a path from $s'$ to $s$ in $h_G(T)$.

**Proof.** The second part of the path correspondence lemma Lemma 10 implies that if there exists a path from $(s', j')$ to $(s, j)$ in $h_G^*(T)$, then there is also a path from $s'$ to $s$ in $h_G(T)$.

For the opposite direction, first notice that existence of a path from $(s, j)$ to $(s', j')$ in $h_G^*(T)$ implies by the path correspondence Lemma 10 (point 2) that there is a path from $s$ to $s'$ in $h_G(T)$ as well. Combined with the assumption that there exists a path from $s'$ to $s$, this gives us a cycle $c$ in $h_G(T)$ from $s$ back to itself via $s'$. In the same way as in the proof of the cycle correspondence Lemma 12 one can show that there exists $l \geq 1$ such that the thread $(c', s)$ (which begins in $(s, j)$) is a cycle in $h_G^*(T)$. This cycle has as a prefix the path from $s$ to $s'$, which means that it contains both $(s, j)$ and $(s', j')$. Thus, there exists a path from $(s', j')$ to $(s, j)$.

The search continues in $h_G(T)$ until (a copy of) the acceptance state $s$ is matched (which means that the desired cycle is detected), or all states reachable from $s$ are generated, without detecting a cycle. (In the latter case the control is returned back to $dfs2$.) If a cycle plainly fair with respect to process $i$ is found the algorithm reports a success only if $i = N$ (line 39), otherwise the cycle check
continues with the next process by calling \texttt{dfs2} with process \(i + 1\) and state space copy \(i + 1\) as arguments (line 40).

The correctness of the RWF algorithm is given by the following claim:

**Theorem 7.** Given a LTS \(T\) the algorithm for reduction under weak fairness (RWF) in Fig. 14 when called in \(h(\hat{s})\) reports a cycle iff there exists a reachable fair acceptance cycle in \(T\).

**Proof.** Suppose that the algorithm reports a cycle. From the correctness of the general DFS algorithm (c.f. [1]) and the structure of the RWF algorithm, it is clear that in that case the algorithm has found in each pair of copies of \(h_G\) and \(h^*_G\), i.e., for each \(i \in I\), a plainly fair cycle \(c_i\) which contains \((r, i)\), where \(r\) is some acceptance state in \(h_G(T)\). By Theorem 6 this implies the existence of a fair acceptance cycle in \(T\).

Assume that there exists a fair acceptance cycle in \(T\). Theorem 5 guarantees that there exists a corresponding subtly fair acceptance cycle in \(h_G(T)\), while Theorem 6 implies the existence of the desired cycles \(c_i\) in \(h^*_G(T)\) for all \(i \in I\). We show that one set of such cycles will be detected by the RWF algorithm. Let \(r\) be the acceptance state which is the first in postorder in \(h_G(T)\) (i.e., the first one deleted from \(Stack1\) in the RWF algorithm) such that there exists a subtly fair acceptance cycle \(c\) in \(h_G(T)\) which contains it. We show that when the RWF algorithm starts the cycle set check from \(r\), in each pair of state space copies corresponding to process \(i\) none of the states from the plainly fair cycle \(c_i\) (corresponding to \(c\)) is already entered in the state space. As a consequence, the algorithm will detect \(c_i\) by generating these states. The proof is by contradiction and it is similar to the proof [7, 5] of the standard NDFS algorithm. Assume that the claim is not true, i.e., that the cycle check for some process \(i\) is wrongly truncated because a state \((r'', C1, C2)\) or \((r'', C2)\) has been found which already was in \(States\). Lemma 13 ensures that such a state could not be generated by the current cycle check. Thus the problematic state must have been generated by some previous cycle check, i.e., a previous call of \texttt{dfs2} or \texttt{dfs3}, from some acceptance state \(r'\) in \(h_G(T)\). From the path correspondence results one can see that this is true if and only if there exists a path \(p_1\) from \(r'\) to \(r''\) in \(h_G(T)\). On the other hand, as there is a path \(p_2\) between \(r''\) and \(r\) (which is a part of the cycle \(c\)), this also means that there exists a path from \(r'\) to \(r\) (the concatenation of \(p_1\) and \(p_2\)). Combining this with the fact that \(r''\) is deleted from \(Stack1\) before \(r\), we conclude from the properties of the DFS that \(r\) must have been generated by \texttt{dfs1} before \(r'\). (This means that \(r\) has been on \(Stack1\) when \(r''\) was visited by the nested search from \(r'\).) In graph-theoretic terminology we say that \(r\) is an ancestor of \(r'\). This implies that there exists a path \(p_3\) between \(r\) and \(r'\) (see, for instance, Lemma 1 of [7]) (This path was contained in \(Stack1\) at the moment when \(r''\) has been visited from \(r'\)). Because \(c\) is subtly fair, the concatenation of the paths \(p_1, c\) (starting at \(r''\) via \(r\) and back to \(r''\)), \(p_2\) and \(p_3\) is also a subtly fair acceptance cycle containing \(r'\) (see Fig. 15). This is in contradiction with our choice of \(r\) to be the first acceptance state removed from \(Stack1\) that is in a subtly fair acceptance cycle. \(\Box\)
```plaintext
1 proc dfs1(s)
2   add s to Stack1
3   add \{s,0\} to States
4   for each transition \( (s,a,s') \) do
5     add \{\{s,0\},a,\{h(s'),0\}\} to Transitions
6     if \{h(s'),0\} not in States then dfs1(h(s')) fi
7   od
8   if accepting(s) then seed := s; dfs2(s,1,1) fi
9   delete s from Stack1
10 end

11 proc dfs2(s,C1,C2) /* the nested search */
12   add \{s,C1,C2\} to Stack2
13   add \{s,C1,C2\} to States
14   for each process \( i := 1 \) to \( N \) do
15     nxt := all transitions \( t \) enabled in \( s \) with Pid(t) == \( i \)
16     if nxt == empty then /* process \( i \) is disabled */
17       if C1 == \( i \) and \{h(s),C2\} not in States then dfs3(h(s),C2) fi
18     else
19       for all \( (s,a,s') \) in nxt do
20         if C1 == \( i \) then /* process \( i \) is executed */
21           if \{h(s'),C2\} not in States then dfs3(h(s'),C2) fi
22         else
23           C1' := perm(s',C1)
24           add \{\{s,C1,C2\},a,pi_s,\{h(s'),C1',C2\}\} to Transitions
25           if \{h(s'),C1',C2\} not in States then dfs2(h(s'),C1',C2) fi
26         fi
27       od
28     fi
29   od
30   delete \{s,C1,C2,fl\} from Stack2
31 end

32 proc dfs3(s,C2)
33   add s to Stack3
34   add \{s,C2\} to States
35   for each transition \( (s,a,s') \) do
36     add \{\{s,C2\},a,\{h(s'),C2\}\} to Transitions
37     if \( h(s') == \) seed then
38       if C2 == \( N \) then report cycle
39         else dfs2(seed,C2+1,C2+1) fi
40     else if \{h(s'),C2\} not in States then dfs3(h(s'),C2) fi
41   od
42   delete s from Stack3
43 end

Fig. 14. Reduction under weak fairness (RWF) algorithm.
```
4.7 Complexity of the RWF Algorithm

We compare the complexity of the RWF algorithm with the complexity of the algorithms of Emerson and Sistla (ES95) [11] and Gyuris and Sistla (GS97) [14]. As we already mentioned, these were the only algorithms for combining weak fairness and symmetry that we could find in the literature. GS97 is an improved version of ES95, so we will mainly refer to the former for comparison. The GS97 algorithm is on-the-fly and it is based on the algorithm for finding all maximal strongly connected components from [1].

It was already mentioned that in our algorithms we need to store only states, while transitions are used only in the proofs. Therefore in the sequel we will consider the number of states as the size, denoted as $|T|$, of a given LTS $T$. In our space complexity calculations we assume that the memory needed for the DFS stack is much smaller than the memory which is used for the States. In practice, this is often a reasonable assumption.

A straightforward analysis of the RWF algorithm shows that we need $N$ copies of the threaded LTS $h_G^T(T)$ (in the dfs2 part of the cycle check for each of the $N$ processes) and $N+1$ copies of the annotated reduction $h_G(T)$ – one copy for the first depth first search with dfs1 and $N$ copies for the dfs3 part of the cycle check. From Def. 10 one can see that the size of $h_G^T(T)$ is $O(N \cdot |h_G(T)|)$. This gives us $O((N^2 + N + 1) \cdot |h_G(T)|)$, i.e., $O(N^2 \cdot |h_G(T)|)$ for the space complexity of RWF. As $h_G(T)$ is usually much smaller (by a factor close to $N!$) than the original LTS $T$ the gain in reduction with the RWF algorithm is obvious.

Moreover, in practice the real memory requirements are much smaller, because there is an efficient way of storing the states of the copies of $h_G^T(T)$ and $h_G(T)$. (Recall that we do not have to store transitions.) The storage technique is due to Gerard Holzmann and it is used in the implementation of the weak fairness algorithm used by the model checker Spin. The technique is a generalization of the trick with the two bits described in Section 2.2 for the original NDFS algo-
algorithm. It uses the fact that the copies of each state \( s \) in \( h_G(T) \), \((s,0),\ldots,(s,N)\) differ only in their counter components. This means that we can represent all \( N+1 \) copies by keeping the description that corresponds to \( s \) plus an additional array \( \text{flag} \) of \( N \) bits to differentiate the copies of \( s \). The bit \( \text{flag}[C] \) is set iff the state \( (s,C) \) is in the state space. In this way, instead of \( (N+1) \cdot |s| \) bits, where \( |s| \) is the size of the description of \( s \), we need only \( |s| + N + 1 \) bits for all copies of \( s \). Because in practice \( |s| \gg N+1 \), the memory complexity is virtually reduced to \( O(|h(T)|) \). With a similar reasoning we can show that we need \( |s| + N^2 \) bits to represent all copies of a given state in \( h^*_G(T) \), instead of \( N^2 \cdot |s| \) with the straightforward approach.

GS97 uses several extra data structures of which two integers are essential. For computing of the maximal strongly connected components, the GS97 algorithm has to keep two special unique numbers for each state. Therefore \( 2 \cdot \log |h_G(T)| \) extra bits are needed in the state space description. The above described efficient storage technique is not used, but even if it had been used, in general \( 2N \log |h_G(T)| \) extra bits would have been needed for the two unique numbers for each of the \( N \) copies of the state. Thus, even with this minimal assumed overhead, for systems where \( 2 \cdot \log |h_G(T)| > N \) our algorithm will have shorter description of the state vector. For instance, for \( \log |h_G(T)| = 20 \), i.e. around \( 10^6 \) states even with 39 processes the overhead in the RWF algorithm will be smaller. In practice most of the time we have a much smaller number of processes, while state spaces of \( 10^6 \) are often encountered.

Regarding the time complexity it is assumed that finding a representative can be done in an efficient way, more precisely, in our calculations we assume a constant time. Unfortunately, this is the case only for some special systems and symmetries. In general, no polynomial algorithm is known to compute a canonical representative. There are however efficient heuristics that work reasonably well in practice (c.f. [19,3]). A constant time for finding a representative is also assumed in the complexity calculations for the GS97 algorithm, therefore, this feature does not have any impact on the comparison.

The time complexity of the algorithm is also dictated by the sizes of \( h_G(T) \) and \( h^*_G(T) \), except that we have to take into account also the transitions. From Def. 10 it is clear that each transition \( r_1 \xrightarrow{a,\pi} r_2 \) from \( h_G(T) \) induces at most \( N \) transitions in \( h^*_G(T) \) (one transition from each copy \( (r_1,i) \)). Thus, also with the transitions included in the size of the state space \( |h^*_G(T)| \) is \( O(N \cdot |h_G(T)|) \). By repeating the reasoning for the space complexity we obtain that the time complexity of the RWF algorithm is \( O(N^2 \cdot |h_G(T)|) \). (Recall that we assume that the canonical representatives are computed in constant time.) This is the same complexity as for the GS97 algorithm.

As the GS97 algorithm requires finding maximal strongly connected components, RWF has all the advantages that NDFS has over the maximal strongly connected components. Probably the most important among those is that, unlike GS97, RWF is compatible with the approximative verification techniques like bit-state hashing [15] or hash-compact [24].
Another advantage of the RWF algorithm is more efficient error detection. Intuitively, it is much easier (faster and within less memory) to find an acceptance cycle than to identify a whole maximal strongly connected component which contains an acceptance state. Also with RWF it is easier to reconstruct an execution which leads to the error.

The annotating permutations in the RWF algorithm are generated as a byproduct of the algorithm for finding representatives. For instance, with virtually no time penalty the algorithm from [3] can be extended to also produce the permutation $\pi_s$ for a given state $s$. In GS97 the permutations (as well as several other extra data structures) are saved as a part of the state description. Although it is mentioned in [14] that keeping the permutations is not necessary, because they can be always recalculated, it is unclear if this can be done without a significant time overhead.

5 Conclusion and Future Work

We presented an efficient algorithm for model checking under weak fairness using reduction based on symmetry and proved its correctness. To this end we first discussed an algorithm for model checking under weak fairness. As the second intermediate step we gave an algorithm for model checking with bisimulation preserving reduction without fairness. The correctness of these two algorithms was also proved. Finally, we presented an extension of the theory developed by Emerson and Sistla [11] in order to fit the concept of the nested depth first search.

An important future task is to prove compatibility of our algorithm with other state space reduction techniques. In that way we can use the combination to obtain better state space reduction. The compatibility of partial order with the bisimulation preserving reduction can be shown along the lines of [12]. It will be more challenging to reconcile symmetry reduction and partial order reduction under weak fairness.

Recently we have been extending the model checker Spin with symmetry reduction with encouraging results [3]. As part of this extension, we intend to try both the bisimulation preserving reduction algorithm (MRNDFS) and the reduction under weak fairness (RWF) algorithm in practice.

It is straightforward to show that the algorithm for bisimulation preserving reduction is compatible with the discrete-time extension of Spin from [2]. We conjecture that this is the case also with the RWF algorithm. The RWF algorithm might also be of interest for dense-time systems regarding the problem of solving non-zenoness from [4] that relies on weak fairness.

Acknowledgments. The author would like to thank Dennis Dams for his detailed proofreading, the numerous examples, and the invaluable suggestions, Jos Baeten for the proofreading and the idea for a more elegant proof of Lemma 4, as well as Leszek Holenderski, Gerard Holzmann, and Doron Peled for their helpful comments regarding the questions treated in this chapter.
References

16. G. J. Holzmann, Personal communication.

8. Conclusions
Conclusions

In this chapter we briefly summarize the results of the thesis and give some general directions for future work. For more detailed comments we refer the reader to the concluding sections of each of the chapters.

1 Summary

In the thesis we have addressed the problem of state-space explosion which is often the main obstacle in many practical applications of model-checking. To this end we have presented improvements of several existing state-space reduction techniques for model checking which are based on explicit enumeration of the state space. We focused on enhancing several practical and efficient algorithms for exploring a part of the state space, while still preserving the property which is to be verified. In particular, our research was concentrated on reductions that exploit the independence of the checked property from the execution order of the actions in the system (partial order reduction), and on reductions that take benefit from the system symmetry.

The main contributions of the thesis related to partial order reduction are the following:

– Exploiting system hierarchy for more efficient partial order reduction via process clustering – we showed how one can profit from more refined treatment of the notion of independence between actions (Chapter 2).
– Solving the compatibility of partial order reduction with priority choice and weak fairness in models with rendez-vous communication (Chapter 3).
– Adapting partial order reduction for untimed systems to discrete-time systems, i.e., for the case when time is modeled with integers (Chapter 4).

The contributions related to symmetry based reductions can be summarized as follows:

– Efficient heuristics for finding representatives of equivalence classes for symmetry based reduction (Chapter 6).
– An efficient algorithm for model checking under weak fairness using symmetry reduction based on nested depth first search (Chapter 7).

The success of model checking is mainly due to the fact that it is relatively easily implementable in tools. Almost all obtained theoretical results listed above are practically implemented and the developed prototype implementations are evaluated on case studies. The implementation work is closely related to the model checker Spin. From the practical contributions we mention:

– DT Spin – an extension of Spin with discrete time which features the discrete-time partial order reduction algorithm presented in the thesis (Chapter 4).
– if2pml – a translator from the modeling language IF to Spin’s input language Promela, meant to be the second part of a translator from SDL to Promela (Chapter 5).


The implementations were tested on examples from the literature and industry with promising results. In particular we mention MASCARA – an industrial protocol that combines wireless communication with ATM (Chapter 5).

Apart from being an indicator for the quality of the results and the implementation, the experiments were often an inspiration for new theoretical work. A typical example is the compatibility of partial order reduction with priority choice and fairness in models with rendez-vous communications. The correction of the partial-order reduction algorithm and the proposed enhancements were directly instigated by experimenting with Spin and its discrete-time extension DT Spin, developed in this thesis.

2 Perspectives

There are indeed many directions in which the research of this thesis can be extended. We discuss some of them in general terms.

Beyond Safety Properties and LTL. In the thesis we focused mainly on the verification of safety properties and LTL. It will be interesting to see if the results can be extended to some other types of properties, for instance the ones expressed in CTL and CTL*. It seems that some of the results, like the process clustering concept and the compatibility with priority choice, can be adapted in an obvious way for the latter. For most of the results however, the adaptation will not be trivial. For instance, it is an open question if the weak fairness algorithm (in the context of symmetry reduction) can be upgraded beyond the nested depth-first search cycle detection (Chapters 3 and 7), which will probably be needed to cover CTL and CTL*.

Adding Time. In this thesis we treated partial order reduction for timed systems, in the case where time is modeled by integers. It will be interesting to see to what extent our approach can be adapted to the case where reals are the time model, and, even further, to hybrid systems. Similar questions that naturally arise involve the application of symmetry reduction to timed systems. Although we have done some initial steps in that direction – mainly in the context of the integer model of time – many problems are still open.

Beyond Model-checking. Since the state-space explosion problem occurs in other areas of computer science (like controller synthesis, optimizations, etc.), the enhancement of the state space reduction techniques presented in this thesis may be applicable beyond model checking and verification. In fact, any problem which
relies on state-space exploration and at the same time features some form of independence of the execution order of the transitions and/or symmetry in the system is a good candidate to benefit from the results in the thesis.

Tools and Case Studies. We are planning several improvements of the software written in the framework of this thesis, as well as practical realizations of the results that were not already implemented. Therefore an obvious task will be to apply the implementations to case studies, mostly on real-world examples. As mentioned above, our experience shows that this is a potential source of new research, driven by practical problems.
Samenvatting

Model-checking is een techniek voor het automatisch opsporen van fouten in en de verificatie van hardware en software. De techniek is gebaseerd op het doorzoeken van de globale toestandsruimte van het systeem. Deze toestandsruimte groeit vaak exponentieel met de grootte van de systeembeschrijving. Als gevolg hiervan is een van de voornaamste knelpunten in model-checking de zogenaamde toestandsexplosie. Er bestaan veel aanpakken om met dit probleem om te gaan. We presenteren verbeteringen van sommige bestaande technieken voor reductie van de toestandsruimte die gebaseerd zijn op expliciete enumeratie van die ruimte. We schenken vooral aandacht aan het verbeteren van verscheidene algoritmies die, hoewel ze slechts een deel van de toestandsruimte onderzoeken, nog steeds gegeven een eigenschap kunnen bewijzen of weerleggen. In het bijzonder is ons onderzoek toegesneden bij twee typen reducties. Het eerste type, partiële-ordening (PO) reductie, buit de onafhankelijkheid van acties in het systeem uit. Het tweede type is een klasse van reducties die voordeel halen uit symmetrieëen van het systeem.

De voornaamste bijdragen van dit proefschrift in verband met de partiële ordening reductie zijn de volgende:

– Het gebruik van systeemhiërarchie voor efficiëntere partiële-ordening reductie door klustering van processen – De meeste model-checking technieken beschouwen het model als een platte compositie van processen. We laten zien hoe de reductie kan profiteren van de systeemstructuur door uitbuiting van de hiërarchie in het systeem (Hoofdstuk 2).
– Correcte syntactische criteria om onafhankelijke acties te vinden voor partiële ordening reductie voor systemen met synchroniserende communicaties die gecombineerd zijn met prioriteit-keuze en/of zwakke fairness (Hoofdstuk 3).
– Partiële-ordening reductie voor discrete tijd – We laten zien hoe het algoritme voor partiële ordening reductie zonder tijd aangepast kan worden, in het geval tijd geregistreerd wordt middels gehele getallen (Hoofdstuk 4).

De bijdragen betreffende symmetrie-gebaseerde reducties kunnen als volgt samengevat worden:

– Efficiënte heuristieken voor het vinden van representanten van equivalentieklassen voor symmetrie-gebaseerde reductie (Hoofdstuk 6).
– Een efficiënt algoritme voor model-checking onder zwakke fairness met toestandsruimte reductietechnieken die gebaseerd zijn op symmetrie (Hoofdstuk 7).

Het succes van model-checking is voornamelijk gebaseerd op de relatief gemakkelijke implementatie in software gereedschappen. Bijna alle bovengenoemde theoretische resultaten zijn geïmplementeerd in de praktijk en de ontwikkelde prototype implementaties zijn geëvalueerd in praktijkstudies. Het meeste implementatie werk is gerelateerd aan de model checker Spin. Van de praktische bijdragen in dit document noemen we:
– DT Spin – een uitbreiding van Spin met discrete tijd die het in het proefschrift gepresenteerd discrete-tijd PO reductie algoritme bevat (Hoofdstuk 4).
– if2pml – een vertaler van de modelleertaal IF naar Spins invoertaal Promela, die als het tweede deel van een vertaler van SDL naar Promela bedoeld is (Hoofdstuk 5).
– SymmSpin – een symmetrie-reductie pakket voor Spin, gebaseerd op de heuristiek beschreven in dit proefschrift (Hoofdstuk 6).

De implementaties zijn getest op voorbeelden uit de literatuur en het bedrijfsleven met bemoedigende resultaten. In het bijzonder noemen we MASCARA – een industriële protocol dat draadloze communicatie met ATM combineert (Hoofdstuk 5).

De experimenten zijn niet alleen een aanwijzing voor de kwaliteit van de resultaten en de implementatie, maar ze waren en zijn ook een inspiratie voor nieuw theoretisch werk. Een typisch voorbeeld is de verenigbaarheid van partiële or- deningen reductie met prioriteit-keuze en fairness in modellen met rendez-vous communicatie. De verbetering van het partiële-ordening algoritme was rechtstreeks geïnspireerd door experimenten met Spin en zijn discrete-tijd uitbreiding DT Spin, ontwikkeld in dit proefschrift.
Titles in the IPA Dissertation Series


P.M. Achten. Interactive Functional Programs: Models, Methods, and Implementation. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. Parallel Local Search. Faculty of Mathematics and Computing Science, TUE. 1996-4


H. Doornbos. Reductivity Arguments and Program Construction. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. Functorial Operational Semantics and its Denotational Dual. Faculty of Mathematics and Computer Science, VUA. 1996-9


N.W.A. Arends. A Systems Engineering Specification Formalism. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. Normalisation in Lambda Calculus and its Relation to Type Inference. Faculty of Mathematics and Computing Science, TUE. 1996-12


M.M. Bonsangue. Topological Dualities in Semantics. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. Algorithms for Graphs of Small Treewidth. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. Process-algebraic Transformations in Context. Faculty of Computer Science, UT. 1997-02


C.J. Bloo. Preservation of Termination for Explicit Substitution. Faculty of Mathematics and Computing Science, TUE. 1997-05


F.A.M. van den Beuken. A Functional Approach to Syntax and Typing. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. Ins and Outs in Refusal Testing. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. A Discrete-Event Simulator for Systems Engineering. Faculty of Mechanical Engineering, TUE. 1998-02
J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03


E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01


C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04


J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08


P.R. D’Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11


T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02


J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06


P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events: a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03


S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06


J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chklaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11


D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14