Tomáš Vojnar

Cut-offs and Automata in Formal Verification of Infinite-State Systems

Habilitation thesis

2007

Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Preface

In this habilitation thesis, we discuss two complementary approaches to formal verification of infinite-state systems—namely, the use *cut-offs* and *automata-based symbolic model checking* (especially the so-called regular model checking). The thesis is based on extended versions of multiple conference and journal papers joint into a unified framework and accompanied with a significantly extended overview of other existing approaches.

The presented original results include cut-offs for verification of parameterised networks of processes with shared resources, the approach of abstract regular model checking combining regular model checking with the counterexample-guided abstraction refinement (CEGAR) loop, a proposal of using language inference for regular model checking, techniques for an application of regular model checking to verification of programs manipulating dynamic linked data structures, the approach of abstract regular tree model checking as well as a proposal of a novel class of tree automata with size constraints with applications in verification of programs manipulating balanced tree structures.

Acknowledgement

Above all, I would like to thank my foreign colleagues in close cooperation with whom I obtained the results presented in this thesis, namely prof. Ahmed Bouajjani and dr. Peter Habermehl from LIAFA, Université Paris 7—Denis Diderot/CNRS and dr. Radu Iosif from VERIMAG, Université Joseph Fourier/CNRS/INPG, Grenoble.

I would further like to thank very much prof. Milan Češka, the head of the research group which I am a member of at FIT BUT, for his continuous moral, professional, and financial support.

I also appreciate very much the discussions and support coming from my colleagues at FIT including assoc. prof. Zdena Rábová (who unfortunately passed away in May 2006), dr. Petr Peringer, dr. Radek Kočí, dr. Bohuslav Křena, dr. Vladimír Janoušek, dr. Martin Hrubý, and also the discussions with dr. Tayssir Touili from LIAFA.

Finally, I am deeply indebted to my parents for their great moral support without which I would have never finished this thesis

Over the time, the work presented in this thesis has been supported by a number of grant projects—in particular, the European 5th Framework IST FET project ADVANCE *under the contract number IST-1999-29082, the long-term institutional research projects of the Czech Ministry of Education number CEZ:J22/98:262200012 "Research in Information and Control Systems" and CEZ MSM 0021630528 "Security-Oriented Research in Information Technology", the projects number 102/01/1485, 102/04/0780, 102/03/D211, 102/07/0322, and 102/05/H050 of the Czech Grant Agency, the ACI project Securité Informatique of the French Ministry of Research, and the Czech-French Barrande project number 2-06-27.*

Contents

1 Introduction 1
1.1 Formal Verification 2
1.2 Model Checking and Infinite-State Systems 5
1.3 Contents and Context of the Thesis 7
1.4 Structure of the Thesis 9

2 Cut-offs on Parameterised Networks of Processes 11
2.1 Cut-off Results On Parameterised Process Networks 12
2.1.1 A Single Control Process and Many Identical User Processes 12
2.1.2 Networks of Processes Communicating by Token Passing 13
2.1.3 Process Networks with Disjunctive or Conjunctive Guards 15
2.1.4 Process Networks with Resource Sharing 15
2.1.5 Cache Coherence Protocols 16
2.1.6 Systems with Parameterised-Size Arrays 16
2.2 RTR Families: Parametric Resource Sharing Networks 17
2.2.1 RTR Families 19
2.2.2 The Specification Logic 23
2.2.3 Verification of Finite Behaviour 25
2.2.4 Verification of Fair Behaviour 29
2.2.5 Process Deadlockability 36
2.2.6 RTR Families and Undecidability 37
2.3 A Summary on Cut-offs and RTR Families 39

3 Regular Model Checking 41
3.1 Finite-State Automata and Transducers 42
3.2 Regular Model Checking: The Basic Idea 43
3.3 Verification by Regular Model Checking 46
3.4 Acceleration in Regular Model Checking 48
3.4.1 Acceleration Schemes 48

3.4.2 Quotienting 49
3.4.3 Extrapolation 52
3.5 Abstract Regular Model Checking 53
3.5.1 A Running Example and Some Basic Assumptions 55
3.5.2 The Method of Abstract Regular Model Checking 56
3.5.3 Automata State Equivalences Based on Predicate Languages 59
3.5.4 Automata State Equivalences Based on Finite-Length Languages 63
3.5.5 Experiments with Abstract Regular Model Checking 65
3.6 Inference of Regular Languages 70
3.6.1 Inference of Regular Languages from Complete Training Sets 72
3.6.2 The Model Checking Algorithm 76
3.6.3 Experiments with the Inference-based RMC 79
3.7 Summary and Extensions of RMC 81

4 Regular Model Checking and Programs with Pointers 85
4.1 An Overview of the Existing Approaches 87
4.1.1 PALE 88
4.1.2 TVLA 89
4.1.3 Other Logic-based Approaches 90
4.1.4 Automata-based Approaches 92
4.1.5 Other Approaches 94
4.2 From Programs to Transducers 98
4.2.1 Encoding Stores as Words 99
4.2.2 Lists with Sharing and/or Loops. 100
4.2.3 Encoding Program Statements as Transducers 100
4.3 Specialised Abstract Regular Model Checking 103
4.3.1 Piecewise 0-k Counter Abstractions 103
4.3.2 Closure Abstractions 105
4.4 Applications and Experimental Results 107
4.4.1 Checking Consistency of Working with the Dynamic Memory 107
4.4.2 Checking More Complex Properties 108
4.4.3 The Results of the Experiments 109
4.5 Summary and Further Work 110

5 Regular Tree Model Checking 113
5.1 Tree Automata and Transducers 114
5.2 RTMC: The Idea and Approaches 116
5.2.1 Acceleration in Regular Tree Model Checking 117
5.3 Abstract Regular Tree Model Checking 118
5.3.1 The Framework of Abstract Regular Tree Model Checking 119

5.3.2 Abstractions over Tree Automata 119
5.3.3 Experiments with Abstract Regular Tree Model Checking 122
5.4 Summary and Further Work on Regular Tree Model Checking . 124

6 Tree Automata with Size Constraints 127
6.1 Works Trying To Go Beyond RMC 128
6.2 Tree Automata with Size Constraints 133
6.2.1 A TASC-based Verification Methodology and a Running Example 136
6.2.2 The Notion of TASC 138
6.2.3 Closure and Decidability Properties of TASC 141
6.2.4 Semantics of Tree Updates 147
6.2.5 Representing Sets of Memory Configurations 149
6.2.6 A Case Study: The Red-Black Tree Insertion 156
6.3 Summary and Further Work 159

7 Conclusion 161
7.1 Summary 161
7.2 Future Research Directions 162

References 165

1 Introduction

Computer-based systems play an increasingly important role in our everyday life. Consequently, their failures may cause more and more serious problems implying huge losses of money, or even worse, the health or life of people can be endangered. Moreover, even if the failures of computer-based systems do not cause direct losses to their users, they may cause significant losses to the producers of these systems (due to the loss of confidence in their products). In addition, there is also a rising danger that even if the errors left in computer-based systems will not make them fail by themselves under normal circumstances, they may become weak points that can be abused for an intentional attack on the given system. At the same time, current computer-based systems are also increasingly more complex, and thus there is also more and more space for errors.

Correspondingly, a significant stress is put on the use of various methods of detecting errors in computer-based systems including code inspection, simulation, and testing. Research and development in these areas is constantly quite active leading to new methods and methodologies as, e.g., model-based design, agile testing, or extreme programming. However, these—let us say "traditional"—methods suffer one important deficiency: *they cannot prove a system correct*, i.e., they cannot prove it to be free of errors wrt. some specification. That is why one can also witness a strong and ever rising interest in the development and applications of *formal verification* methods that can remove this constraint. Moreover, it turns out that even though the formal verification process may sometimes not be completely finished due to its high computational price, it may still be quite valuable. This is because before the process runs out of resources, it may find a number of errors that are often different from those find by traditional methods, which is due to the different principles on which these methods work.

The interest of various leading industrial companies and organisations in formal verification methods may be documented by the existence of research groups specialised in this area, e.g., within Microsoft, Intel, IBM, Siemens, NASA, Airbus, etc., and/or in a strong cooperation of the companies with the academia on this subject. Another indication is also the support coming from

the companies to various leading scientific conferences from the given area like TACAS or CAV. The interest in formal verification methods is, of course, not limited to industrial companies, but appears within various open source projects too—we can mention here, for instance, the attempts to formally verify the L4 microkernel.

Motivated by the above, the work presented in this thesis contributes to the research on certain forms of formal verification. We now first discuss a bit more the notion and various existing forms of formal verification and then we introduce the concrete contents of the thesis and position it within the research on formal verification.

1.1 Formal Verification

We use the term *formal verification* to denote verification methods based on formal, mathematical roots and (at least potentially) capable of proving error freeness of systems wrt. some correctness specification. The potential to detect all errors wrt. a given specification is called *soundness* of a method. It means that if such a method terminates and claims a system correct wrt. a certain specification, the system is really correct.[1] On the other hand, we call a method *complete* if it does not raise false alarms, i.e., if it does not report spurious errors.

The probably most popular approaches used in computer-aided formal verification include theorem proving, model checking, static analysis, and abstract interpretation. We have to, however, note that the meaning of some of these terms is sometimes not completely sharp and not always understood in exactly the same way. Also the approaches may be used in various combinations and/or non-standard extensions.

The present thesis deals, in particular, with the area of model checking. However, for a better orientation in the subject, we now very briefly introduce all the mentioned approaches and characterise their relations.

Theorem Proving

Theorem proving is an (often only semi-automated) approach using some inference system for deducing various theorems about the examined system from the facts known about the system and from general theorems of logic, arithmetics, etc. This approach is quite close to classical mathematical reasoning, just it is supported by computer-aided tools, the so-called theorem provers (e.g., PVS [OSRSC01], Isabelle [NPW05], ACL2 [KMe00b, KMe00a],

[1] Note that sometimes the potential of a method to be sound may be deliberately sacrificed to its efficiency leading to an error detection method with formal verification roots (cf., e.g., the bit-state hashing method used, for instance, in the Spin model checker [Hol97] where different reachable states of a system being verified are not distinguished when they have the same hash value).

and many others), taking care of remembering all of the so-far deduced facts, of correctly applying inference rules, etc. The approach is very general, but often very hard to use. The approach is sometimes also weak in generating counterexamples (diagnostic information) to correctness specifications in faulty systems—one may have troubles to distinguish whether the effort to prove some property is failing because there is an error in the system being examined, or because the user of the method is not bright enough. On the other hand, there is also a lot of progress in developing theorem provers capable of running in a fully automated mode, which are recently often used in a combination with other methods—for instance, as a support for different kinds of automated abstraction (such as the predicate abstraction [GS97]) for model checking.

Model Checking

Model checking [CGP99] is an approach of automated checking whether a model of a system (where the model can sometimes be identical to the system) satisfies a certain correctness specification. The specification is typically written in some temporal logic like LTL [Pnu77], CTL [CE81], CTL* [EH86], or μ-calculus [Koz83], but some simpler specification means such as the C-like assertions or end-state or progress labels known from Promela [Hol97] can also be used. Traditionally, model checking is based on a systematic exploration of the state space of the examined model. Its roots can be traced back to the works [CE81, QS82]. Model checking can usually be fully automated and can generate error traces explaining why a certain property does not hold in a given system. Its main disadvantage is the *state space explosion* problem, i.e., the need to cope with the exponential growth of the number of reachable states in the size of the examined systems.

To cope with the state explosion problem, many different heuristics have been proposed [Val98, CGP99]. Among them, without trying to give a complete account of the numerous existing techniques, we can mention *symbolic verification* dealing in an efficient way with sets of states instead of the individual states. The most famous symbolic verification method is probably the one based on *binary decision diagrams* [Bry86, BCM$^+$92], which is behind many of the successes of model checking, especially in hardware.

Further, various *state space reduction techniques* have been proposed. Some of them are based, for instance, on symmetries [CFJ93, ID96a, SMG97] or partial-order reduction techniques [Val88, KP88, God91]. These techniques allow one to avoid generating and exploring some of the states as it is clear that their properties are not important in general, or at least wrt. the property being checked, or are covered by the properties of other states.[2] Moreover,

[2] For instance, symmetries allow one to claim some states equal by exchanging the roles of some of their components—one can, e.g., rotate the philosophers in the well-known dining philosophers problem, etc. Partial-order reduction techniques allow one to explore only some interleavings of concurrently enabled actions as it can be shown that nothing new can be seen in the other interleavings.

when the property to be examined is being checked in parallel to the state space generation, which is denoted as the so-called *on-the-fly* model checking, one can avoid not only the generation of some not important states (which can be seen not to have any chance to influence the given property of interest), but also stop as soon as an error is found without having to generate many further reachable states.

The capabilities of formal verification based on model checking can also be enhanced by combining model checking with *modular verification* [Pnu89, CLM89, CCST05, AMN05] or *automated abstraction* [CGL94, GS97, BLO98, CGJ+00b, CCG+04, HJMS03]. In the latter case, traditionally, model checking is applied to systematically and precisely explore the state space of an abstract model derived from the concrete model to be verified.[3] The precision of the abstraction may be adjusted on demand when a spurious counterexample caused by a too rough abstraction is encountered. The refinement can be driven by the spurious counterexample itself leading to the so-called *counterexample-guided abstraction refinement* (*CEGAR*) framework. A prominent role among automated abstraction techniques applied with model checking is currently played by the so-called *predicate abstraction*[4] [GS97], which is very often used especially in the domain of software verification [HJMS03, CCG+04].

Static Analysis

Static analysis concentrates on intelligent browsing through the source code of a system and collecting some abstract (approximate) information about it rather than systematically exploring its reachable states. There exist many different forms of static analysis ranging from simple syntactic checks and type analyses to more complicated fixpoint computations on the control flow graph of the system being examined (cf., e.g., [EM04, Sch06]). In many cases, static analysis is not designed primarily for checking correctness of programs, but to be used within compiling, code optimisation, etc. Static analyses are often highly specialised. On the other hand, they sometimes just collect some information about the system, and it is up to the user to exploit it for a given verification task.

Compared to model checking, static analyses have often the advantage of not needing any model of the environment in which the system should run and of being able to handle very big code bases. The need to model

[3] Moreover, recently, abstraction is beginning to be used also within the state space exploration process itself to speed it up [McM05] or to make it terminate in infinite-state model checking [BHV04].

[4] Very roughly said, within predicate abstraction, one does not track the precise values of various state variables, but only some predicates about them (e.g., $x \geq 0$, $x \leq y + 5$, etc.). To find out how the validity of the tracked predicates changes in response to the transitions being fired, one can use specialised decision procedures or theorem provers operated in a fully automated way

the environment and usually also parts of the system being examined (which would otherwise be too big to be handled) may hurt the model checking approach by being quite expensive and also by possibly hiding some errors (that may be ruled out by the manual modelling done) [EM04].

On the other hand, not tracking the (exact) values that particular system variables may get, can lead to a vast number of false alarms raised by static analysis.[5] Moreover, some kinds of errors may be difficult or impossible to discover via certain static analyses. For instance, it may be difficult or impossible to identify all possible "syntactic patterns" that could lead to certain errors, and then the otherwise very efficient methods like those mentioned in [EM04] may be hard to use.

Of course, there is a number of works trying to increase the precision and expressiveness of static analysis by remembering more and more about the reachable values of the variables. Then, however, these approaches are getting closer and closer to model checking both in their advantages and disadvantages.

Abstract Interpretation

Abstract interpretation is introduced in [CC77, CC79, Cou81, Deu92] as a theory of a sound approximation of the semantics of computer programs intended mainly for constructing various static analyses. Abstract interpretation consists in giving a certain class of programs several semantics linked by abstraction and concretisation functions. The semantics are based on monotonic functions over ordered sets, typically lattices. The abstraction, concretisation, and semantic functions must be linked in a certain way prescribed by the abstract interpretation framework (we will not go into the formal details here—they can be found in the above referenced literature). The semantics of a program is computed as the least fixpoint of the semantic function over the given ordered set. To make the computation terminate even over infinite semantic domains, widening functions are to be provided.

The notion of abstract interpretation is quite flexible and can be instantiated in a number of ways significantly differing in their preciseness (basically ranging from the preciseness of simple syntactic static analyses to full model checking). Abstract interpretation is also sometimes used as a formal framework in which abstractions to be used together with model checking are defined (like in the case of predicate abstraction [GS97, BPR01]).

1.2 Model Checking and Infinite-State Systems

In theorem proving, static analysis, and abstract interpretation, dealing with infinite-state systems is traditional. On the other hand, most of the research

[5] Sometimes, the tools developed in this area ignore much of the potential errors detected in order not to overwhelm the user. Then, however, the approach becomes unsound—though it may still be very valuable.

on model checking has so-far concentrated on systems with possibly large, but finite state spaces. However, infinite-state systems are also quite common in practice. Infinity can arise due to dealing with various kinds of *unbounded data structures* such as

- push-down stacks needed for dealing with recursive procedures,
- FIFO (and other kinds) of queues of waiting processes or messages (in the latter case, it is perhaps more natural to speak about channels),
- unrestricted counters (or integer variables), or
- dynamic linked data structures (such as lists or trees), etc.

Another source of infinity may be dealing with *time* or other *continuous variables*, which arise when analysing continuous or hybrid systems. Finally, a need to deal with infinite state spaces may arise also due to various kinds of *parameters* (such as the maximum value of some variable, the maximum length of a queue, or the number of processes in a system) when one wants to verify the given system for any value of the parameters. In the last case, to be more precise, we are dealing with *infinite families* of systems (which themselves may be finite-state or infinite-state). Nevertheless, the need to verify the system for any member of the family leads anyway to infinite-state verification as the union of the state spaces of all the family members is infinite.

Consequently, techniques applying model checking in the area of infinite-state systems have begun to appear as well. Moreover, there also appear various combinations, mutual influence, and inspiration between model checking approaches and theorem proving, static analysis, and abstract interpretation.

Approaches to Model Checking of Infinite-State Systems

One possibility of verifying infinite-state systems via model checking is to use the so-called *cut-offs*. Cut-offs are such bounds on the various infinite resources that when one successfully verifies a given system up to the cut-off bounds, the verified properties are guaranteed to hold even when the bounds are removed. Cut-offs are one of the techniques that we discuss in more detail in this work. Basically, if one can find a suitable cut-off, infinite-state verification may be reduced to a finite-state one.

Another possibility is using various kinds of (finite-range) *abstractions*. The abstractions considered in the literature range from predicate abstraction [GS97] (indeed, when we use a finite number of Boolean predicates, the abstract state space becomes finite regardless of the original domains of the concrete state variables) to various specialised abstractions proposed, for instance, for verifying parameterised networks of processes [ID96b, BBLS00, PXZ02].

Further, one may use *symbolic model checking* based on some kind of a finite representation of infinite sets of states by means of logics, automata, grammars, etc. Among successful symbolic verification methods, we can count the so-called *regular model checking*, which is another of the approaches that

we study in detail in the following. Regular model checking is based on representing infinite, but regular sets of states by finite-state automata. Its advantages are that it is (usually) fully automated and quite generic. Another successful symbolic verification is then, for instance, the symbolic model checking approach based on the so-called zones [Dil89, HNSY94] that has turned out to be very successful in the domain of model checking real-time systems modelled by timed automata [AD94, HNSY94].

Yet another group of often studied approaches is based on various ways of *automated induction*—cf., e.g., [WL89, KM95, MQS00, LHR97, CR00, PRZ01]. Many of these works use the so-called *network invariants* which provide an abstraction of a composition of any number of processes. It then suffices to use model checking to verify that the behaviour of a single process is covered by the network invariant, a composition of the network invariant and a process is also covered by the invariant, and the invariant satisfies the given property of interest.

Decidability Issues

Of course, when dealing with infinite-state systems, one very quickly reaches *undecidability* of the verification problems. The same naturally holds for parametric verification [AK86]. Therefore, most of the model checking methods proposed in this area are either *not fully automated*, or they are *semi-algorithmic* heuristics, i.e., they do not guarantee termination, or they allow an indefinite answer of the type "don't know" to be returned. However, even such techniques may often prove quite successful on many practical examples.

Moreover, not all of the problems are undecidable. *Decidability* has been proved, e.g., for model checking of the full modal μ-calculus over *push-down systems* [Wal96, BS97] (for a fixed LTL formula, the problem is even polynomial [BEM97, FWW97, EHRS00]). Reachability, safety, inevitability, and (fair) termination are decidable over *lossy FIFO channel systems* [Fin94, CFI96b, AJ96a, MS02]. None of them, however, is decidable in a primitive recursive time [Sch02a] (and some other verification problems, including recurrent reachability, and hence, liveness, are undecidable [AJ96b]). Further, many verification problems of *timed automata* are decidable due to the region equivalence [AD94, ACD93] (with reachability checking being PSPACE-complete in this domain). There are also a number of positive decidability results for various *dynamic networks of concurrent processes with recursion* [May00a, BMOT05], and so on. Even in such cases, it may, however, be sometimes more advantageous to use semi-algorithmic heuristic approaches that may turn out to deliver more efficient results in practice.

1.3 Contents and Context of the Thesis

In this work, we discuss two of the above mentioned approaches—namely, the *use of cut-offs* and the *regular model checking* method. Into these two

areas, we have concentrated most of our recent research. We present (some of) the results that we obtained in this area together with an overview of other existing proposals.

An interesting point to note about the two techniques we discuss is that they are quite complementary: Cut-off results are often specialised and may allow one to transform a particular infinite-state verification problem to a finite-state one. On the other hand, regular model checking is generic and usable even if no reduction to finite-state model checking can be done.

As a part of our original contribution, we, in particular, present a series of *cut-off results for verification of parameterised networks of processes with shared resources* [BHV02, BHV03]. Our contribution further includes a proposal of a combination of the regular model checking approach with automated abstraction yielding the so-called *abstract regular model checking* technique [BHV04]. This framework is generalised to dealing with *tree languages* [BHRV05, BHRV06a] as well. Further, we have proposed, and we present here, an original technique of using abstract regular model checking for *verification of programs with dynamic linked data structures* [BHMV05]. As an alternative to abstract regular model checking, we have also studied an original *application of language inference methods in regular model checking* [HV04, HV05]. Finally, as one of the existing techniques trying to go beyond the use of regular languages, we discuss our proposal of a new class of *tree automata with size constraints* and its application to the verification of programs manipulating balanced tree structures [HIV05, HIV06].

In order not to make the thesis too broad, we have chosen not to discuss in detail all of our recent works despite that some of them are also related to verification of infinite-state systems (and, in particular, various kinds of symbolic verification) [EV05, ČEV05, ČEV06, BBH$^+$06a, BHRV06b]. We, however, briefly discuss these approaches in the appropriate parts of the thesis.

Most of the original results that we present here were achieved in a tight and fruitful *cooperation with our foreign partners*. In particular, we acknowledge our cooperation with Ahmed Bouajjani and Peter Habermehl from LIAFA, Université Paris 7—Denis Diderot/CNRS and Radu Iosif from VERIMAG, Université Joseph Fourier/CNRS/INPG, Grenoble. Moreover, several *Ph.D. students* have participated on the work including Pierre Moro from LIAFA and Adam Rogalewicz, Pavel Erlebach, and Aleš Smrčka (with whom we are, e.g., currently trying to apply abstract regular model checking to a parameterised verification of some hardware components) from FIT. The three latter students are supervised by prof. M. Češka while the author of the thesis is their co-supervisor.

The thesis is based on extended versions of papers originally published at renowned conferences and in journals. The results are accompanied with a significantly extended discussion of other existing works and presented in a unified framework.

1.4 Structure of the Thesis

In the following chapters, we always start with an introduction that is more specialised to the technique being presented than the general introduction presented here. Then, we explain the basic principles of the presented technique and give an overview of the results existing in the given area. Subsequently, we present in detail our original results followed by some concluding remarks specific for the appropriate area.

In particular, in Chapter 2, we discuss the cut-off technique and our results in this area. Chapter 3 speaks about regular model checking with a stress put on abstract regular model checking and the use of language inference for regular model checking. Further, in Chapter 4, we discuss the area of verifying programs with dynamic linked data structures and our proposal of using (abstract) regular model checking in this area. In Chapter 5, we discuss a generalisation of (abstract) regular model checking to (abstract) regular tree model checking. Finally, in Chapter 6, we touch upon the existing approaches trying to use more than regular languages for symbolic model checking and we describe our class of tree automata extended with size constraints and their use for verification of programs manipulating balanced tree structures.

We close the thesis by a general summary and a future work overview in Chapter 7.

2

Cut-offs on Parameterised Networks of Processes

The technique of finding *cut-offs* is one of the approaches often successfully used for verification of parameterised systems (and/or systems with some sort of unbounded resources like the length of communication queues, the amount of available dynamic memory, the size of counters, etc.). Given a class of properties and a class of systems to be checked, *the principle of the cut-off technique* is to provide a bound on the involved parameters (unbounded resources) such that to check the given property over the given system (or a family of systems in the case of parameterisation), it suffices to check the property in a setting where the parameters (originally unbounded resources) are bound to a certain finite value (or a finite set of values).

An advantage of the cut-off approach is that if one manages to come up with a cut-off in some setting (especially when the cut-off happens to be small or moderate), it allows one to turn an infinite-state verification problem to a finite-state one. Then, one can re-use some of the various efficient tools already developed for verification of finite-state systems instead of using the usually much heavier methods available for a direct work with infinite-state systems. On the other hand, a disadvantage of the cut-off technique is that the cut-offs are usually very specific to a particular class of systems and properties, and it is not easy to adapt them for other settings. Thus, we can say that when one is facing a parameterised and/or infinite-state verification problem, it is worthwhile trying to reduce it to a finite-state one via the cut-off technique. If this is not successful (or the cut-off obtained is too big), one can then proceed to the use of other methods.

Note that the cut-off technique may also be used when model checking a finite, but large system by reducing the number of processes (or resources) to deal with.

In the literature, the term a "*small model*" is sometimes used instead of speaking about a cut-off. The technique is not discussed primarily only in the framework of formal verification but also in other contexts. For example, it is sometimes used when providing a decision procedure for a logic with variables ranging over some unbounded domain (as, for instance, in [PRSS02, EVW97]).

Such results can then often be used back in the area of formal verification of infinite-state or parameterised systems.

Here, we concentrate on the direct use of cut-offs for formal verification of parameterised networks of processes. Note, however, that even in this case, the cut-off is not always expressed directly on the number of processes to be considered. Sometimes, the cut-off is phrased in terms of the maximum length of a behaviour to be considered or the maximum number of verification steps to be performed. In such cases, the number of processes may be limited implicitly, e.g., by starting with zero processes and counting the addition of a process as one step [GS92], or it may be completely abstracted away by remembering only whether at least one process is at some control location or not [EN96]. Having a cut-off with the same size phrased on the number of verification steps is of course better than having it phrased on the number of processes. This is because in the latter case, one has still to generate and explore the state space of the system with the given number of processes which is usually exponential in the number of processes.

Below, we first summarise some of the best known results obtained in the area of using cut-offs for verifying parameterised networks of processes. Subsequently, we explain in detail our contribution to this area. Our original results concentrate on verification of parameterised networks of processes with sets of shared resources being available via certain FIFO-based access policies. These results were first published in [BHV02, BHV03], and the full version of the paper is currently submitted to a journal [BHV06].

2.1 Cut-off Results On Parameterised Process Networks

2.1.1 A Single Control Process and Many Identical User Processes

An early work using cut-offs for automatic verification of parameterised networks of processes is [GS92]. In this paper, the basic considered model describes families of systems with two kinds of processes: *a single finite-state "control" process* and *an arbitrary number of identical finite-state "user" processes.* The processes perform two kinds of actions: either "silent" internal transitions or *CCS-like synchronised transitions.* The synchronisation is based on having a finite set of communication symbols Σ and a set of their complements $\overline{\Sigma}$. Two processes then synchronise when one of them performs a c transition whereas the other one performs a $\overline{c}$ transition (here, $c \in \Sigma$ and $\overline{c} \in \overline{\Sigma}$ are complementary symbols). The paper provides a doubly-exponential decision procedure for verification of LTL properties of single processes (i.e., speaking either about the control process or a user process), which is based on a cut-off on the length of the behaviours to be considered. The result can be used for verifying mutual exclusion too.

The paper then also considers various modifications of this basic setting. For verification of LTL\X properties[1] of single processes on fair runs, a technique using reachability of vector addition systems is used. A modification of this technique can be used for checking deadlockability as well. An algorithm based on integer linear programming, which is polynomial in the size of process descriptions, is then shown for the case of having only an arbitrary number of identical user processes (perhaps structured to some classes but with no distinguished unique process). On the other hand, it is shown that the verification problem for the basic setting described in the previous paragraph becomes undecidable when one allows process quantification in LTL formulae (i.e., when allowing to specify that certain propositions should hold for certain processes described via quantified process variables instead of always referring to the properties of a single process—we will formally define such a logic later on).

In [EN96], families of systems consisting again from *a single finite-state control process and an arbitrary number of identical user finite-state processes* are considered. Unlike in [GS92], the processes *run synchronously*—in every global transition, every process fires its local transition, and the processes *communicate via some sort of sharing of information about their states.* In particular, transitions of processes have guards of the form $\exists p.\varphi(p)$ where φ is a boolean combination of propositions over the state of the control process and over the state of the p-th user process. (The model was used to describe and verify, e.g., a certain bus arbitration protocol.) The work proposes an exponential-time cut-off-based procedure for verifying tree kinds of properties: (1) $A\varphi$ and $E\varphi$ where φ is an LTL formula over the states of the control process, (2) $\forall p.A\varphi(p)$ and $\forall p.E\varphi(p)$ where φ is an LTL formula over the states of the control process and the states of the p-th user process, and (3) $\forall p_1 \neq p_2.A\varphi(p_1, p_2)$ and $\forall p_1 \neq p_2.E\varphi(p_1, p_2)$ where φ is an LTL formula over the states of the control process and the states of the p_1-th and p_2-th user processes[2]. The result uses a cut-off on the length of behaviours to be explored in an abstraction of the considered systems where one only distinguishes whether none or at least one process is at a certain control location. Some optimisations of the result are then available for deterministic user processes and for checking safety properties.

2.1.2 Networks of Processes Communicating by Token Passing

Another work which employs cut-offs for verification of parameterised networks of processes is [EN95]. In particular, *ring networks of identical finite-state processes* are considered. Among the processes a special token is being

[1] LTL\X denotes LTL without the *next-time* operator X. The exclusion of this operator is quite common when using cut-off results. The reason is that X allows one to—in some sense—count the processes.

[2] Temporal logics that allow us to explicitly speak about the local states of certain processes are often called *indexed temporal logics* [ES97].

passed in a fixed direction that allows the processes to perform certain critical actions. The transitions of the processes are thus divided into ones that can be performed only with the token and ones that can be performed even without it. In addition, processes have transitions allowing them to receive and send the token from/to their neighbours. The paper considers a version of CTL$^*\setminus$X with process quantification and obtains the following small cut-off results (moreover, their proofs may further be used as an inspiration for obtaining cut-offs for other similar properties if need be):

- Rings of size 2 suffice for verifying properties of the form $\forall p.\varphi(p)$ where $\varphi(p)$ is a CTL$^*\setminus$X formula with p being a process variable (no further variables and quantification are allowed). An example of such a formula is, e.g., the eventual access property $\forall p.AG\ (request(p) \Rightarrow AF\ using(p))$.
- Rings of size 3 suffice for verifying properties of the form $\forall p.\varphi(p, p+1)$.
- Rings of size 4 suffice for verifying properties of the form $\forall p_1 \neq p_2.\varphi(p_1, p_2)$. An example of such a formula is, e.g., the mutual exclusion $\forall p_1 \neq p_2.AG\neg(critical(p_1) \wedge critical(p_2))$.
- Rings of size 5 suffice for verifying properties of the form $\forall p_1 \neq p_2.\varphi(p_1, p_1+1, p_2)$.

The token considered above does not carry any additional information. When it is allowed to carry a value and change the value any number of times, the model becomes Turing powerful. The case when the value can change only a finite number of times is considered in [EK04]. For a certain form of the communicating processes (that are in some sense deterministic, but that need not be homogeneous) and for the unidirectional circulation of the tokens (note that there may be multiple circulating tokens here), a cut-off $b(|T_i| + |T_j|)$ is obtained for properties of the form $A\varphi(p_i, p_j)$ or $E\varphi(p_i, p_j)$ where p_i, p_j are two chosen processes whose local states are monitored by the LTL$\setminus$X formula φ. Here, b is the maximum number of times that a token may change its value, and T_i, T_j are sets of tokens that may be seized by processes p_i, p_j. The result may be generalised for the case when there is a FIFO queue for tokens waiting in a process for handling, as well as for the case when a token may change the direction of its circulation, but only when its value changes. A much more involved cut-off is then shown even for the case of a general bidirectional communication using a single token whose value can change a finite number of times. The work is motivated by applications, e.g., in verification of distributed algorithms (like the leader election) where often a circular topology is used and the messages being exchanged change only a finite number of times.

Another recent work that explores possibilities of using cut-offs in *networks of processes communicating by token passing* is [CTTV04]. In particular, the work considers networks of homogeneous finite-state processes that are *interconnected in an arbitrary oriented graph.* The processes have token-independent and token-dependent transitions. The latter can be fired only if a process posses the only token circulating in the network. The token can be

sent and received along the arcs in the network topology, and it is supposed that every processes infinitely many times receives and releases the token. It is then proved that model checking of any k-indexed LTL\X property over an arbitrarily large network of an arbitrary topology can be reduced to model-checking at most $3^{k.(k-1)}2^k$ networks with at most $2k$ processes (for the important class of formulae with two tracked processes, this means verifying up to 36 networks with up to 4 processes). The result gives the same cut-off for families of networks with some topology, though it is not effective from the point of view of saying how to find the particular networks to verify. The work further shows that an analogical cut-off cannot be found for indexed CTL\X.

2.1.3 Process Networks with Disjunctive or Conjunctive Guards

The work [EK00] considers parameterised networks of *finite-state processes which are classified into a finite number of classes.* The processes communicate via *disjunctive or conjunctive guards over the states of the present processes.* A disjunctive guard allows a process to test whether there is a process which is in some local control state. A conjunctive guard allows a process to test whether all processes are in some of selected local states. Three kinds of properties are considered: (1) for all processes p of a single class, $A\varphi(p)$ or $E\varphi(p)$, (2) for all processes $p_1 \neq p_2$ of a single class, $A\varphi(p_1, p_2)$ or $E\varphi(p_1, p_2)$, and (3) for any process p_1 of one class and any process p_2 of another class, $A\varphi(p_1, p_2)$ or $E\varphi(p_1, p_2)$. Here, φ is an LTL\X formula speaking about local states of the quantified processes. The work obtains cut-offs requiring at most $|P_c|+3$ or $2|P_c|+1$ processes from every class c to be taken into account ($|P_c|$ is the size of the finite control of processes from class c) for disjunctive or conjunctive guards, respectively. For properties ranging over all paths for disjunctive guards and for properties ranging only on finite or only infinite paths in the case of conjunctive guards, the cut-offs are optimised to requiring only a small, fixed number of processes (up to three) from each class (sometimes only two or three processes are needed in total) yielding a really efficient, polynomial time verification procedure.

2.1.4 Process Networks with Resource Sharing

In [EK02], the authors consider *networks of heterogeneous finite-state processes communicating via competing for resources shared between pairs of processes.* The processes always perform some resource-independent steps, then acquire certain resources, perform some resource-dependent transitions, release all the resources, and repeat this behaviour. All of the resources attached to a process must always be acquired, they must be acquired wrt. a certain partial order on the resources, and once some resources are acquired, they cannot be released before all resources are acquired. The work shows a cut-off down to two processes for safety properties of the form $E_{fin}/A_{fin}\varphi(p_1, p_2)$ where E_{fin}/A_{fin} range over finite paths and φ is an LTL\X formula for fixed

processes p_1, p_2. A cut-off down to three processes is shown for liveness properties over unconditionally fair paths (where every process must keep running forever) for a fixed pair of processes and LTL\X formulae. For the case of ring-shaped networks, a cut-off down to (at most) five processes is then shown for proving deadlockability (whereas the general case is shown to be NP-complete).

In Section 2.2, we present in detail our own results obtained in the area of verifying parameterised process networks with shared resources. We, however, consider a different class of systems (with resources shared by all processes, with a prioritised FIFO access to resources, etc.) as well as a bit different class of properties (including liveness under weak/strong fairness).

Another result that relies partially on cut-offs is then [KIG05] where verification of an arbitrary number of *concurrent, recursive threads communicating via locking shared resources is considered*. The threads are modelled as push-down systems with a finite control. It is shown that for single-index LTL\X properties over finite as well as over infinite behaviours, it suffices to consider a single thread. For doubly indexed properties, dealing with two threads is shown sufficient. As the threads are push-down systems, the obtained verification problem for two tracked processes is still undecidable in general. However, it is shown to be decidable for the case of having only nested locks (where it suffices to explore two augmented threads in isolation and then combine the obtained results).

2.1.5 Cache Coherence Protocols

In [EK03], a set of methods is presented for formal verification of cache coherence protocols allowing them to be proved correct for a parametric number of caches. A cut-off result is presented for one of the classes of such protocols, namely the so-called *invalidation-based snoopy protocols*, i.e., protocols that on every write operation invalidate the written block in all caches other the one in which the write operation happened. The paper presents a cut-off down to seven processes (caches) for verification of properties of the form $\forall p_1 \neq p_2.A\ \varphi(p_1, p_2)$ and $\forall p_1 \neq p_2.E\varphi(p_1, p_2)$ where φ is an LTL\X formula over the local states of p_1, p_2, and E/A range either over all finite behaviours, over all behaviours, or over all unconditionally fair behaviours.

2.1.6 Systems with Parameterised-Size Arrays

An interesting work on the verification of parameterised systems that also relies (at least partially) on cut-offs is [APR$^+$01]. The work builds upon the deductive verification approach where one tries to find an inductive invariant φ such that φ is implied by the initial condition of the considered system, φ is inductive (i.e., if it holds before a transition of the system, it holds after the transition too), and φ implies the property to be checked. The authors provide a heuristic to automatically derive the inductive invariant (based on

generalising the behaviour of a bounded-size system), and they provide a cut-off result allowing one to automatically check all the described verification conditions.

The considered systems are described by *a finite number of boolean variables, parameterised-size integers, and arrays with parametric bounds containing either boolean values, parameterised-size integers, or other parameterised arrays.* (A parameterised number of processes may easily be encoded via a parameterised array containing the local states of the particular processes.) The property to be checked is supposed to be an invariant of the form $\forall \overline{x}.\psi(\overline{x})$. The synthesised inductive invariant has the same form. The transition relation is of the form $\exists \overline{y}\ \forall \overline{x}.\rho(\overline{y}, \overline{x})$. The atomic formulae allowed are correctly-typed comparisons between variables and array references (the framework, moreover, allows one to encode "+1" and "+1 *mod N*" constraints too). The work proves a cut-off for checking validity of the considered formulae, which is polynomial in the number of variables in the formulae and in the system.

2.2 RTR Families: Parametric Resource Sharing Networks

We now present our own results from the area of using cut-offs for verification of parameterised systems [BHV02, BHV03, BHV06]. In particular, the results concern verification of *parameterised networks of processes with resource sharing.*

Managing concurrent access to shared resources is a fundamental problem that appears in many contexts, e.g., operating systems, multithreaded programs, control software, etc. The critical properties to ensure are typically (1) mutual exclusion when exclusive access is required, (2) absence of starvation (a process that requires a resource will eventually get it), and (3) absence of deadlocks.

Many different instances of the above problem can be defined depending on the assumptions on the allowed actions for access to resources and the policies for managing the access to these resources.

In our work, we consider systems with a *finite number of resources shared by a parametric number of identical processes.* These processes can require a set of resources, get access and use the requested resources, and release the used resources. The requests can be of a low-priority or a high-priority level. The access to the resources is managed by a *locker* according to a *FIFO-based policy taking into account the priorities of the requests*—i.e., a waiting high-priority request can overtake waiting low-priority ones. As a special case allowing for an optimised treatment, we then examine the situation when no high-priority requests are used, and the locker behaves according to the pure FIFO discipline.

For an abstract description of the concerned systems, we define a model based on extended automata with queues recording the identities of the waiting processes for each resource. Then, we address the verification problem

for families of such systems with an arbitrary number of processes (called *RTR families* where RTR stands for request-take-release) against formulae of the temporal logic *LTL*$\backslash$*X* extended with *global process quantification.* We consider two interpretation domains for the logic: the set of finite behaviours (which is natural for safety properties), and the set of fair behaviours (in order to cover liveness properties). In addition, we consider the parametric verification problem of process deadlockability too. Thus, we cover all the three most important classes of properties for the given application domain.

The *two* obstacles involved in the considered systems (parameterisation and having multiple queues over an unbounded domain of process identifiers) complicate the use of any of the known methods for verification of infinite-state and/or parameterised systems. Using cut-offs appears to be the easiest approach here. When establishing our cut-off results, we consider the question whether it is possible to find cut-off bounds that do not depend on the structure of the involved processes and the formula at hand, but only on the number of resources and the number of processes quantified in the formula. Indeed, these numbers are relatively small, especially in comparison to the size of the process control automata.

We show that for RTR families where the *pure FIFO resource management* is used (i.e., no high-priority access to resources is required), parametric verification of finite as well as fair behaviour is decidable against all LTL$\backslash$X formulae with global process quantification. The cut-off bound in the finite behaviour case is the number of quantified processes, whereas it is this number plus the number of resources in the fair behaviour case. These bounds lead to practical finite-state verification. Furthermore, we show that the verification of process deadlockability is decidable too (with the cut-off bound being equal to the number of resources).

On the other hand, for the case of dealing with RTR families that *distinguish low-priority and high-priority requests*, we show that—unfortunately—structure-independent cut-offs do not exist in general neither for the interpretation of the considered logic on finite nor fair behaviours. However, we show that even for such families, parametric verification of finite behaviour is decidable, e.g., against reachability/invariance formulae, and parametric verification of fair behaviour is decidable against formulae with a single quantified process. In this way, we cover, e.g., verification of the (for the given application domain) key properties of mutual exclusion and absence of starvation. For the former case, we even obtain a structure-independent cut-off equal again to the number of quantified processes. For verification of fair behaviour against single process formulae, no general structure-independent cut-off can be found, but we provide a structure-dependent one, and in addition, we determine a significant subclass of RTR families where a structure-independent cut-off for this particular kind of properties does exist. Finally, we show that process deadlockability can be solved in the case of general RTR families via the same (structure-independent) cut-off as in the case of the families not using high-priority requests.

Further, we show that although the queues in RTR families are not communication queues, but just waiting queues, and the above decidability results may be established, the model is still quite powerful, and decidability may easily be lost when trying to deal with a bit more complex properties to verify. We illustrate this by proving that parametric finite-behaviour verification becomes *undecidable* (even for families not using high-priority requests) for LTL\X extended with the notion of *local process quantification* [ES97], which allows one to examine different processes in different encountered states.

Let us note that the work we present here was originally motivated by an interest of Ericsson in having a more complete verification method for verifying the use of shared resources in some of their ATM switches than just using finite-state model checking to verify some isolated instances of the involved parameterised verification problems as in [AED02]. However, the operations for access to shared resources and the resource management policies that we consider are quite natural in general in concurrent applications dealing with shared resources.

Out of the other cut-off approaches presented in Section 2.1, the closest one to our result is [EK02]. However, both the system model and the employed proof techniques differ. As we have said in the previous section, the processes in [EK02] need not be identical, the number of resources is not bounded, but, on the other hand, only two fixed processes may compete for a given resource, and their requests are served in a random order (there are no FIFO queues in [EK02]). Moreover, some of the properties to be verified we consider here are different from [EK02] (e.g., we deal with the more realistic notion of weak/strong fairness compared to the unconditional one used there, etc.).

Outline of the rest of the section: Below, we first formalise the notion of RTR families and define the specification logic we use. Then, we present our cut-off results for finite and fair behaviour and process deadlockability as well as the undecidability result. We mostly provide proof ideas of the presented results only—the complete proofs can be found in the extended version of [BHV03] that is available over the Internet.

2.2.1 RTR Families

The Model of RTR Families

Processes in systems of RTR families are controlled by *RTR automata*. An RTR automaton over a finite set of resources is a finite automaton with the following kinds of actions joint with transitions: skip (denoted by τ—an abstract step not changing resource utilisation), request and, when it is the turn, take a set of resources at the low- or high-priority level (`rqt`/`prqt`), and, finally, release a set of resources (`rel`).

Let us, however, stress that we allow processes to block inside (`p`)`rqt` transitions[3] while waiting for the requested resources to be available for them.

[3] We use (`p`)`rqt` when addressing both `rqt` as well as `prqt` transitions.

Therefore, a single (p)rqt transition in a model semantically corresponds to two transitions, which we denote as (p)req (request a set of resources) and (p)take (start using the requested resources when enabled to do so by the locking policy).

Definition 2.2.1 *An* RTR automaton *is a 4-tuple* $\mathcal{A} = (R, Q, q_0, T)$ *where* R *is a set of resources,* Q *is a set of control locations,* $q_0 \in Q$ *is an initial control location, and* $T \subseteq Q \times A \times Q$ *is a transition relation over the set of actions* $A = \{\tau\} \cup \{a(R') \mid a \in \{\mathtt{rqt}, \mathtt{prqt}, \mathtt{rel}\} \wedge R' \neq \emptyset \wedge R' \subseteq R\}$*. The sets* R*,* Q*,* T*, and* A *are nonempty, finite, pairwise disjoint, and disjoint with* $\mathbb{N}$*.*

An *RTR family* $\mathcal{F}(\mathcal{A})$ over an RTR automaton $\mathcal{A}$ is a set of *systems* S_n consisting of $n \geq 1$ identical processes controlled by $\mathcal{A}$ and identified by elements of $P_n = \{1, ..., n\}$. (In the following, if no confusion is possible, we usually drop the reference to $\mathcal{A}$.) We denote as *RTR*$\backslash$*P families* the special case of RTR families whose control automata contain no high-priority request actions.

Configurations

For the rest of the section, let us suppose working with an arbitrary fixed RTR family $\mathcal{F}$ over an automaton $\mathcal{A} = (R, Q, q_0, T)$ and with a system $S_n \in \mathcal{F}$.

To make the semantics of RTR families reflect the fact that processes may block in (p)rqt actions, we extend the set Q of “explicit” control locations to Q_t containing a unique internal control location q_t for each transition $t \in T$ based on a (p)rqt action. Furthermore, let T_t be the set obtained from T by preserving all τ and rel transitions and splitting each transition $t = (q_1, (\mathtt{p})\mathtt{rqt}(R'), q_2) \in T$ to two transitions $t_1 = (q_1, (\mathtt{p})\mathtt{req}(R'), q_t)$ and $t_2 = (q_t, (\mathtt{p})\mathtt{take}(R'), q_2)$.

We define the *resource queue alphabet* of S_n as $\Sigma_n = \{s(p) \mid s \in \{\mathtt{r}, \mathtt{pr}, \mathtt{g}, \mathtt{u}\} \wedge p \in P_n\}$. The meaning is that a process has requested a resource in the low- or high-priority way, it has been granted the resource, or it is already using the resource. A *configuration* c of S_n is then a function $c : (P_n \to Q_t) \cup (R \to \Sigma_n^*)$ that assigns the current control locations to processes and the current content of queues of requests to resources. Let C_n be the set of all such configurations.

Resource Granting and Transition Firing

We now introduce the *locker function* Λ implementing the considered FIFO resource management policy with low- and high-priority requests. This function is to be applied over configurations changed by adding/removing some requests to/from some queues in order to grant all the requests that can be granted wrt. the given strategy in the given situation. Note that in the case of

RTR\P families, the resource management policy can be considered the pure FIFO policy.

A high-priority request is granted iff none of the needed resources is in use by or granted to any process, nor it is subject to any sooner raised, but not yet granted, high-priority request. A low-priority request is granted iff the needed resources are not in use nor granted and they are not subject to any sooner raised request nor any later raised high-priority request that can be granted at the given moment. (High-priority requests that currently cannot be granted do not block sooner raised low-priority requests.) Formally, for $c \in C_n$, we define $\Lambda(c)$ to be a configuration of C_n equal to c up to the following for each $r \in R$:

1. If $c(r) = w_1.\mathtt{pr}(p).w_2$ for some $p \in P_n$, $w_1, w_2 \in \Sigma_n^*$ s.t. $c(p) = q_t$ for a certain $t = (q_1, \mathtt{prqt}(R'), q_2) \in T$ and for all $r' \in R'$, $c(r') = w'_1.\mathtt{pr}(p).w'_2$ with $w'_1 \in \{\mathtt{r}(p') \mid p' \in P_n\}^*$ and $w'_2 \in \Sigma_n^*$, we set $\Lambda(c)(r)$ to $\mathtt{g}(p).w_1.w_2$. (Intuitively, `pr` queue items can overtake `r` items if no `pr` item of the given request is preceded by a `u`, `g`, or `pr` item.)
2. If $c(r) = \mathtt{r}(p).w$ for some $p \in P_n$, $w \in \Sigma_n^*$ s.t. $c(p) = q_t$ for a certain $t = (q_1, \mathtt{rqt}(R'), q_2) \in T$ and for all $r' \in R'$, $c(r') = \mathtt{r}(p).w'$ with $w' \in \Sigma_n^*$, and the premise of case 1 is not satisfied for r', we set $\Lambda(c)(r)$ to $\mathtt{g}(p).w$. (All `r` items of a low-priority request to be granted must be the heads of the appropriate queues and cannot be followed by any `pr` items of high-priority requests that can be granted.)

We define *enabling* and *firing of transitions* in processes of S_n via a predicate $en \subseteq C_n \times T_t \times P_n$ and a function $to : C_n \times T_t \times P_n \to C_n$.

For all transitions $t = (q_1, \tau, q_2) \in T_t$ and $t = (q_1, a(R'), q_2) \in T_t$, $a \in \{\mathtt{rel}, \mathtt{req}, \mathtt{preq}\}$, we define $en(c, t, p) \Leftrightarrow c(p) = q_1$. For each transition $t = (q_1, (\mathtt{p})\mathtt{take}(R'), q_2) \in T_t$, we define $en(c, t, p) \Leftrightarrow c(p) = q_1 \wedge \forall r \in R'\ \exists w \in \Sigma_n^* : c(r) = \mathtt{g}(p).w$. Intuitively, a transition is enabled in some process if the process is at the source control location of the transition and, in the case of (p)take, if the appropriate request has been granted.

Firing of a transition $t = (q_1, \tau, q_2) \in T_t$ simply changes the control location mapping of p from q_1 to q_2, i.e., $to(c, t, p) = (c \setminus \{(p, q_1)\}) \cup \{(p, q_2)\}$.

Firing of a (p)req transition t corresponds to registering the request in the queues of all the involved resources and going to the internal waiting location of t. The locker is applied to (if possible) immediately grant the request. For $t = (q_1, (\mathtt{p})\mathtt{req}(R'), q_2) \in T_t$, we define $to(c, t, p) = \Lambda((c \setminus c^-) \cup c^+)$ where $c^- = \{(p, q_1)\} \cup \{(r, c(r)) \mid r \in R'\}$ and $c^+ = \{(p, q_2)\} \cup \{(r, c(r).(\mathtt{p})\mathtt{r}(p)) \mid r \in R'\}$.

For a transition $t = (q_1, (\mathtt{p})\mathtt{take}(R'), q_2) \in T_t$, we simply change all the appropriate `g` queue items to `u` items and finish the concerned (p)rqt transition, i.e., $to(c, t, p) = (c \setminus c^-) \cup c^+$ with c^- as in the case of (p)req and $c^+ = \{(p, q_2)\} \cup \{(r, \mathtt{u}(p).w) \mid r \in R' \wedge c(r) = \mathtt{g}(p).w\}$.

Finally, a `rel` transition removes the head `u` items from the queues of the given resources provided they are really owned by the given process. The locker is applied to grant all the requests that may become unblocked. Formally, for

a transition $t = (q_1, \texttt{rel}(R'), q_2) \in T_t$, we fix $to(c,t,p) = \Lambda((c \setminus c^-) \cup c^+)$ with $c^- = \{(p,q_1)\} \cup \{(r,c(r)) \mid r \in R' \wedge \exists w \in \Sigma_n^* : c(r) = \texttt{u}(p).w\}$ and $c^+ = \{(p,q_2)\} \cup \{(r,w) \mid r \in R' \wedge w \in \Sigma_n^* \wedge c(r) = \texttt{u}(p).w\}$.

Suppose now that a process p_1 is requesting a set of resources R' in a configuration $c \in C_n$, i.e., $c(p_1) = q_2$ for some $(q_1, (\texttt{p})\texttt{req}(R'), q_2) \in T_t$. We say that p_1 and its current request are *blocked* by a process p_2 on a resource $r \in R'$ in c iff $c(r) \in \Sigma_n^*.s_2(p_2).\Sigma_n^*.s_1(p_1).\Sigma_n^*$ and $s_1 = \texttt{pr} \Rightarrow s_2 \neq \texttt{r}$. In other words, p_1 is blocked by p_2 on r iff p_2 has a suitable item in the queue of r that precedes some queue item of p_1 in this queue. Low-priority requests may be blocked by queue items of any type, in the case of high-priority requests, $\texttt{r}$ items are not powerful enough.

Behaviour of Systems of RTR Families

Let S_n be a system of an RTR family $\mathcal{F}$. We define the *initial configuration* c_0 of S_n to be such that $\forall p \in P_n : c_0(p) = q_0$ and $\forall r \in R : c_0(r) = \epsilon$. By a *finite behaviour* of S_n starting from $c_1 \in C_n$, we understand a sequence $c_1(p_1,t_1)c_2...(p_l,t_l)c_{l+1}$ such that for each $i \in \{1,...,l\}$, $en(c_i,t_i,p_i)$ holds, and $c_{i+1} = to(c_i,t_i,p_i)$. If c_1 is the initial configuration c_0, we may drop a reference to it and speak simply about a finite behaviour of S_n. The notion of *infinite behaviours* of S_n can be defined in an analogous way. A *complete behaviour* is then either infinite or such that it cannot be extended any more.

We say a complete behaviour is *weakly (process) fair* iff each process that is eventually always enabled to fire some transitions, always eventually fires some transitions. More formally, we say a complete behaviour β_n is weakly fair iff for each $p \in P_n$, if β_n is infinite, and $\exists i : \forall j \geq i : \exists t \in T_t : en(c_j,t,p)$ holds, then $\forall i : \exists j \geq i : \exists t \in T_t : c_{j+1} = to(c_j,t,p)$ holds too.

We may call a complete behaviour *strongly (process) fair* iff each process that is always eventually enabled to fire some transitions, always eventually fires some transitions. Formally, a complete behaviour β_n is strongly fair iff for each $p \in P_n$, if β_n is infinite, and $\forall i : \exists j \geq i : \exists t \in T_t : en(c_j,t,p)$ holds, then $\forall i : \exists j \geq i : \exists t \in T_t : c_{j+1} = to(c_j,t,p)$ holds too. However, we do not deal with strong fairness in the following because in our model, the notions of strong and weak fairness coincide.

Lemma 2.2.1 *A complete behaviour of a system S_n of an RTR family $\mathcal{F}$ is strongly fair iff it is weakly fair.*

Proof. (Idea) Due to the separation of requesting resources and starting to use them and the impossibility of cancelling once issued grants of resources, a process cannot temporarily have no enabled transitions without firing any transitions. □

For a behaviour $\beta_n = c_1(p_1,t_1)c_2(p_2,t_2)...$ of a system S_n of an RTR family $\mathcal{F}$, we call the configuration sequence $\pi_n = c_1c_2...$ a *path* of S_n corresponding

to β_n and the transition firing sequence $\rho_n = (p_1, t_1)(p_2, t_2)...$ a *run* of S_n corresponding to β_n. If the behaviour is not important, we do not mention it. We denote Π_n^{fin}, $\Pi_n^{fin} \subseteq C_n^+$, the set of all finite paths of S_n and Π_n^{wf}, $\Pi_n^{wf} \subseteq C_n^+ \cup C_n^\omega$, the set of all paths of S_n corresponding to complete, weakly fair behaviours.

2.2.2 The Specification Logic

We concentrate (with the exception of process deadlockability) on verification of process-oriented, linear-time properties of systems of RTR families. For specifying the properties, we use the below described extension of LTL\X, which we denote as *MPTL* (i.e., temporal logic of many processes). As is often the case when using cut-offs, we exclude the next-time operator from our framework to avoid its ability to count processes.

We extend LTL\X by *global process quantification*[4] in a way inspired by ICTL* (see, e.g., [ES97]) and allowing us to easily reason over systems composed of a parametric number of identical processes. We also allow for an explicit distinction whether a property should hold for all paths or for at least one path out of a given set. Therefore, we introduce a single top-level *path quantifier* to our formulae. We restrict quantification in the following way: (1) We implicitly require all variables to always refer to distinct processes. (2) We allow only uniformly universal (or uniformly existential) process and path quantification.

Finally, we limit *atomic formulae* to testing the current control locations of processes. We allow for referring to the internal control locations of request transitions too, which corresponds to asking whether a process has requested some resources, but has not become their user yet.

The Syntax of MPTL

Let PV, $PV \cap \mathbb{N} = \emptyset$, be a set of process variables. We first define the syntax of *MPTL path subformulae*, which we build from atomic formulae $at(p, q)$ using boolean connectives and the until operator. For $V \subseteq PV$ and $p \in V$, we have:

$$\varphi(V) ::= at(p, q) \mid \neg\varphi(V) \mid \varphi(V) \vee \varphi(V) \mid \varphi(V)\ \mathcal{U}\ \varphi(V)$$

As syntactical sugar, we can then introduce in the usual way formulae like `tt`, `ff`, $\varphi(V) \wedge \varphi(V)$, $\Box\varphi(V)$, or $\Diamond\varphi(V)$.

Subsequently, we define the syntax of *universal* and *existential MPTL formulae*, which extend MPTL path subformulae by process and path quantification used in a uniformly universal or existential way. For $V \subseteq PV$, we have:

$$\Phi_a ::= \forall V : A\ \varphi(V) \qquad \Phi_e ::= \exists V : E\ \varphi(V)$$

[4] Later, we extend MPTL by local process quantification and show that this leads to undecidability. We do not introduce local process quantification immediately in order not to complicate the presentation of the positive decidability results.

In the rest of the section, we commonly specify sets of quantified variables by listing their elements in some chosen order. Using MPTL formulae, we can then express, for example, *mutual exclusion* as $\forall p_1, p_2 : A \,\Box\, \neg(at(p_1, cs) \wedge at(p_2, cs))$ or *absence of starvation* as $\forall p : A \,\Box\, (at(p, req) \Rightarrow \Diamond\, at(p, use))$.

The Formal Semantics of MPTL

Suppose working with a set of process variables PV. As we require process quantifiers to always speak about distinct processes, we call a function $\nu_n : PV \rightarrow P_n$ a *valuation* of PV iff it is an injection.

Suppose further that we have a system S_n of an RTR family $\mathcal{F}$. Let $\pi_n \in C_n^* \cup C_n^\omega$ denote a (finite or infinite) path of S_n. For a finite (or infinite) path $\pi_n = c_1 c_2 ... c_{|\pi_n|}$ ($\pi_n = c_1 c_2 ...$), let π_n^l denote the suffix $c_l c_{l+1} ... c_{|\pi_n|}$ ($c_l c_{l+1} ...$) of π_n, respectively. (For a finite π_n with $|\pi_n| < l$, $\pi_n^l = \epsilon$.)

Given a path π_n of S_n and a valuation ν_n of PV, we inductively define the semantics of MPTL path subformulae $\varphi(V)$ as follows:

- $\pi_n, \nu_n \models at(p, q)$ iff $\pi_n = c.\pi_n'$ and $c(\nu_n(p)) = q$.
- $\pi_n, \nu_n \models \neg\varphi(V)$ iff $\pi_n, \nu_n \not\models \varphi(V)$.
- $\pi_n, \nu_n \models \varphi_1(V) \vee \varphi_2(V)$ iff $\pi_n, \nu_n \models \varphi_1(V)$ or $\pi_n, \nu_n \models \varphi_2(V)$.
- $\pi_n, \nu_n \models \varphi_1(V)\ \mathcal{U}\ \varphi_2(V)$ iff there is $l \geq 1$ such that $\pi_n^l, \nu_n \models \varphi_2(V)$ and for each k, $1 \leq k < l$, $\pi_n^k, \nu_n \models \varphi_1(V)$.

As for any given behaviour β_n of S_n, there is a unique path π_n corresponding to it, we will also sometimes say in the following that β_n satisfies or unsatisfies a formula φ meaning that π_n satisfies or unsatisfies φ. We will call the processes assigned to some process variables by ν_n as processes *visible* in π_n via ν_n.

Next, let $\Pi_n \subseteq C_n^* \cup C_n^\omega$ denote any set of paths of S_n. (Later we concentrate on sets of paths corresponding to all finite or fair behaviours.) We define the semantics of MPTL universal and existential formulae as follows:

- $\Pi_n \models \forall V : A\ \varphi(V)$ iff for all valuations ν_n of PV and all $\pi_n \in \Pi_n$, $\pi_n, \nu_n \models \varphi(V)$.
- $\Pi_n \models \exists V : E\varphi(V)$ iff $\pi_n, \nu_n \models \varphi(V)$ for some PV valuation ν_n and some $\pi_n \in \Pi_n$.

Evaluating MPTL over Systems and Families

Let S_n be a system of an RTR family $\mathcal{F}$. Given a universal or existential MPTL formula Φ, we say the finite behaviour of S_n satisfies Φ, which we denote by $S_n \models_{fin} \Phi$, iff $\Pi_n^{fin} \models \Phi$ holds. We say the weakly fair behaviour of S_n satisfies Φ, which we denote by $S_n \models_{wf} \Phi$, iff $\Pi_n^{wf} \models \Phi$ holds.

Next, we introduce a notion of MPTL formulae satisfaction over RTR families, in which we allow for specifying the minimum size of the systems to

be considered.[5] We go on with the chosen uniformity of quantification and for a universal MPTL formula Φ_a, an RTR family $\mathcal{F}$, and a lower bound l on the number of processes to be considered, we define $\mathcal{F}, l \models^a_{fin} \Phi_a$ to hold iff $S_n \models_{fin} \Phi_a$ holds for *all* systems $S_n \in \mathcal{F}$ with $l \leq n$. Dually, for an existential MPTL formula Φ_e, we define $\mathcal{F}, l \models^e_{fin} \Phi_e$ to hold iff $S_n \models_{fin} \Phi_e$ holds for *some* system $S_n \in \mathcal{F}$ with at least l processes. We suppose the same notions of MPTL formulae satisfaction over families to be introduced for weakly fair behaviour too.

2.2.3 Verification of Finite Behaviour

As we have already indicated, one of the problems we examine in our work is verification of finite behaviour of systems of RTR families against correctness requirements expressed in MPTL. In particular, we concentrate on the *parametric finite-behaviour verification problem* of checking whether $\mathcal{F}, l \models^a_{fin} \Phi_a$ holds for a certain RTR family $\mathcal{F}$, a universal MPTL formula Φ_a, and a lower bound l on the number of processes to be considered. The problem of checking whether $\mathcal{F}, l \models^e_{fin} \Phi_e$ holds for a certain existential MPTL formula Φ_e is dual, and we will not cover it explicitly in the following.

A Cut-Off Result for RTR\P Families

We first examine the parametric finite-behaviour verification problem for the case of RTR\P families. Let $\Phi_a \equiv \forall p_1, ..., p_k : A\ \varphi(p_1, ..., p_k)$ be a universal MPTL formula with k globally quantified process variables. We show that for any RTR\P family $\mathcal{F}$, the problem of checking $\mathcal{F}, l \models^a_{fin} \Phi_a$ can be reduced to a simple finite-state examination of the system $S_k \in \mathcal{F}$ with k processes. At the same time, the processes to be monitored via $p_1, ..., p_k$ may be fixed to $1, ..., k$. We denote the resulting verification problem as checking whether $S_k \models_{fin} A\ \varphi(1, ..., k)$ holds. Consequently, we can say that, e.g., to verify mutual exclusion in an RTR\P family $\mathcal{F}$, it suffices to verify it for processes 1 and 2 in the system of $\mathcal{F}$ with only these two processes.

Below, we first give a basic cut-off lemma and then we generalise it to the above.

Lemma 2.2.2 *For any given RTR\P family $\mathcal{F}$ and any given MPTL path formula $\varphi(p_1, ..., p_k)$, the following holds for systems of $\mathcal{F}$:*

$$\forall n \geq k : S_n \models_{fin} \forall p_1, ..., p_k : A\ \varphi(p_1, ..., p_k) \Leftrightarrow S_k \models_{fin} A\ \varphi(1, ..., k)$$

[5] Fixing the maximum size would lead to finite-state verification. Although our results could still be used to simplify such verification, we do not discuss this case here.

Proof. (Sketch) ($\Rightarrow$) We convert a counterexample behaviour of S_k to one of S_n by adding some processes and letting them idle at q_0. ($\Leftarrow$) To reduce a counterexample behaviour of S_n to one of S_k, we remove the invisible processes and the transitions fired by them (these processes may be shown to only restrict the behaviour of others by blocking some resources, and so their removal does not disable any of the remaining transitions) and we permute the processes to make $1, ..., k$ visible (all processes are initially equal and their names are not significant). □

By using Lemma 2.2.2 and properties of MPTL, we now easily obtain the above promised result.

Theorem 2.2.1 *Let $\mathcal{F}$ be an RTR$\backslash$P family and let $\Phi_a \equiv \forall p_1, ..., p_k : A\ \varphi(p_1, ..., p_k)$ be an MPTL formula. Then, checking whether $\mathcal{F}, l \models^a_{fin} \Phi_a$ holds is equal to checking whether $S_k \models_{fin} A\ \varphi(1, ..., k)$ holds.*

Proof. First, for all $S_n \in \mathcal{F}$ with $l \leq n < k$, $S_n \models_{fin} \Phi_a$ trivially holds as there are not k distinct processes here. If $S_k \models_{fin} A\ \varphi(1, ..., k)$ holds, Lemma 2.2.2 implies $S_n \models_{fin} \Phi_a$ holds for all $k \leq n$, which is what is needed or even more (for $k < l$). If $S_n \models_{fin} \Phi_a$ holds for $l \leq n$, $S_k \models_{fin} A\ \varphi(1, ..., k)$ follows either directly for $l \leq k$ or from Lemma 2.2.2. □

Inexistence of Structure-Independent Cut-Offs for RTR Families

Unfortunately, as we prove below, for families with prioritised resource management, the same reduction as above cannot be achieved even when we allow the bound to also depend on the number of available resources and fix the minimum considered number of processes to one.

Theorem 2.2.2 *For MPTL formulae Φ_a with k process variables and RTR families $\mathcal{F}$ with m resources, the parametric finite-behaviour verification problem of checking whether $\mathcal{F}, 1 \models^a_{fin} \Phi_a$ holds is not, in general, decidable by examining just the systems $S_1, ..., S_n \in \mathcal{F}$ with n being a function of k and/or m only.*

Proof. (Idea) In the given framework, we can check whether in some system of the RTR family $\mathcal{F}$ based on the automaton from Fig. 2.1, some process p_1 can request `A`,`B` before some process p_2 requests `B`, but the wish of p_2 is granted before that of p_1. As shown in Fig. 2.1, the above happens in $S_n \in \mathcal{F}$ with $n \geq 3$, but not in $S_2 \in \mathcal{F}$ (the overtaking between visible processes `2` and `3` is impossible without invisible process `1`). Moreover, when we start extending the `B` and `AB` branches by more and more pairs of the appropriate (`p`)`rqt`/`rel` actions without extending the `A` branch, we exclude more than one process

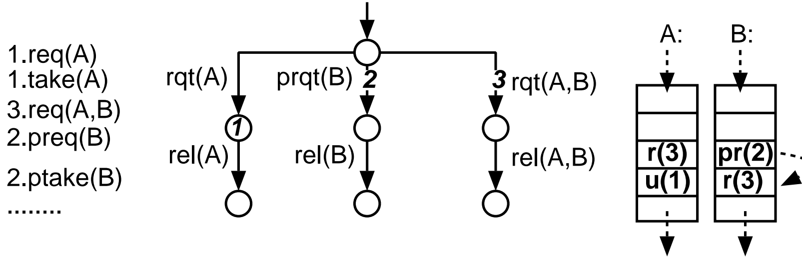


Fig. 2.1. A scenario problematic for the application of cut-offs (the run from the left is visualised on the RTR automaton and the appropriate resource queues)

to run in these branches via adding `rqt(C)`/`rel(C)` (`rqt(D)`/`rel(D)`) at their beginnings and ends, and we ask whether p_1 and p_2 can exhibit more and more overtaking, we will need more and more auxiliary processes in the `A` branch although k and m will not change. □

Despite the above result, there is still some hope that the parametric finite-behaviour verification problem for RTR and MPTL can be reduced to finite-state model checking. Then, however, the bound on the number of processes would have to also reflect the structure of the RTR automaton of the given family and/or the structure of the formula being examined. We leave the problem in its general form open for future research. Instead, we show below that for certain important subclasses of MPTL, the number of processes to be considered in parametric finite-behaviour verification can be fixed to the number of process variables in the formula at hand as in the RTR\P case (although the underlying proof construction is more complex). In this way, we cover, among others, mutual exclusion as one of the key properties of the considered class of systems.

Cut-Offs for Subclasses of MPTL

The first subclass of MPTL formulae that we consider is the class of *invariance* and *reachability* formulae of the form

$$\Psi_a ::= \forall V : A\ \Box\psi(V) \qquad \Psi_e ::= \exists V : E\ \Diamond\psi(V)$$

in which $\psi(V)$ is a boolean combination of atomic formulae $at(p, q)$. Mutual exclusion is an example of a property that falls into this class.

Let $\Psi_a \equiv \forall p_1, ..., p_k : A\ \Box\psi(p_1, ..., p_k)$ be an arbitrary invariance MPTL formula with k quantified process variables. We show that for any RTR family $\mathcal{F}$, the parametric problem of checking $\mathcal{F}, l \models^a_{fin} \Psi_a$ can be reduced to the finite-state problem of verifying $S_k \models_{fin} A\ \Box\psi(1, ..., k)$ with the number of processes fixed to k and the processes to be monitored via $p_1, ..., p_k$ fixed to

$1, ..., k$. As above, we first state a basic cut-off lemma, which we subsequently generalise.

Lemma 2.2.3 *For any RTR family $\mathcal{F}$ and any non-temporal MPTL path formula $\psi(p_1, ..., p_k)$, the following holds for systems of $\mathcal{F}$:*

$$\forall n \geq k : S_n \models_{fin} \forall p_1, ..., p_k : A \,\Box\psi(p_1, ..., p_k) \Leftrightarrow S_k \models_{fin} A \,\Box\psi(1, ..., k)$$

Proof. (Sketch) The case of ($\Rightarrow$) can be treated like in Lemma 2.2.2. In the ($\Leftarrow$) case, to reduce a counterexample behaviour of S_n to one of S_k, we first leave out all invisible processes and the transitions fired by them and permute the processes to make $1, ..., k$ visible (again like in Lemma 2.2.2). This is, however, not sufficient. The obtained transition sequence may not be firable because some overtaking among visible processes (a high-priority request is granted before a sooner issued low-priority one) may be possible only with the help of some invisible process, which blocks the low-priority request (cf. Fig. 2.1).

We solve the above problem in such a way that we replace overtaking among processes by postponed firing of their requests. More precisely, we postpone firing of (`p`)`req` transitions to be just before firing of the corresponding (`p`)`take` transitions (or at the very end of the run if the appropriate (`p`)`take` transition is not fired). Then, since the preserved processes release resources as they used to and they do not block them by requests before all originally overtaking requests are served, it can be shown that the firability of the reduced transition sequence is guaranteed. Moreover, the behaviour is modified in a way invisible for a reachability formula (negation of $\Box\psi$), which ensures that we obtain the desired counterexample in S_k. □

Theorem 2.2.3 *Let $\mathcal{F}$ be an RTR family and let Ψ_a be an invariance MPTL formula of the form $\Psi_a \equiv \forall p_1, ..., p_k : A \,\Box\psi(p_1, ..., p_k)$. Then, checking whether $\mathcal{F}, l \models^a_{fin} \Psi_a$ holds is equal to checking whether $S_k \models_{fin} A \,\Box\psi(1, ..., k)$ holds.*

Proof. Similar to the proof of Theorem 2.2.1 with the use of Lemma 2.2.2 replaced by Lemma 2.2.3. □

We now discuss yet another subclass of MPTL that can be handled within parametric finite-behaviour verification of RTR in the same way as above. This time, we allow any of the MPTL operators to be used, but we exclude distinguishing whether a process is at a location from which it can request some resources or whether it has already requested them. In other words, we allow only the MPTL formulae for which whenever there is a transition $t = (q_1, (\mathtt{p})\mathtt{rqt}(R'), q_2) \in T$, we never use $at(p, q_1)$ or $at(p, q_t)$, but at most $(at(p, q_1) \vee at(p, q_t))$. Let us denote such *location-restricted* MPTL formulae

by Υ_a/Υ_e and their path subformulae by υ. Using such formulae, we can, for example, check whether some overtaking among the involved processes is possible or excluded (though not on the level of particular requests).

Theorem 2.2.4 *Let $\mathcal{F}$ be an RTR family and let Υ_a be a location-restricted MPTL formula of the form $\Upsilon_a \equiv \forall p_1, ..., p_k : A\ \upsilon(p_1, ..., p_k)$. Then, checking whether $\mathcal{F}, l \models^a_{fin} \Upsilon_a$ holds is equal to checking whether $S_k \models_{fin} A\ \upsilon(1, ..., k)$ holds.*

Proof. (Sketch) Similar to Lemma 2.2.3 and Theorem 2.2.3 when we take into account that the postponed firing of requests manifests itself just by some processes staying longer at locations q_1 before going to q_t for some $t = (q_1, (\mathtt{p})\mathtt{rqt}(R'), q_2) \in T$, which cannot be distinguished by $(at(p, q_1) \vee at(p, q_t))$. □

2.2.4 Verification of Fair Behaviour

We next discuss verification of fair behaviour of systems of RTR families against correctness requirements expressed in MPTL. The results presented in this section can be applied for verification of liveness properties, such as absence of starvation, of systems of RTR families. As for finite-behaviour verification, we consider the *problem of parametric verification of weakly fair behaviour*, i.e., checking whether $\mathcal{F}, l \models^a_{wf} \Phi_a$ holds for an RTR family $\mathcal{F}$, a universal MPTL formula Φ_a, and a lower bound l on the number of processes.

We show first that under the pure FIFO resource management, considering up to $m + k$ processes—with m being the number of resources and k the number of visible processes—suffices for parametric verification of weakly fair behaviour against any MPTL formulae. By contrast, for the prioritised resource management, we prove that (as in the case of finite behaviour verification) there does not exist any general, structure-independent cut-off that would allow us to reduce parametric verification of weakly fair behaviour to finite-state verification. Moreover, we show that, unfortunately, the inexistence of a structure-independent cut-off concerns, among others, also verification of the very important property of absence of starvation. Thus, for the needs of parametric verification of fair behaviour, we subsequently examine in more detail the possibility only sketched in the previous section, i.e., trying to find a cut-off reflecting the structure of the appropriate RTR automaton and/or the structure of the formula.

A Cut-Off Result for RTR\P Families

Let $\mathcal{F}$ be an RTR\P family with m resources and $\Phi_a \equiv \forall p_1, ..., p_k : A\ \varphi(p_1, ..., p_k)$ a universal MPTL formula with k process variables. We show

that the parametric verification problem of weakly fair behaviour for $\mathcal{F}$ and Φ_a can be reduced to a series of finite-state verification tasks in which we do not have to examine any systems of $\mathcal{F}$ with more than $m+k$ processes. The processes to be monitored via $p_1, ..., p_k$ may again be fixed to $1, ..., k$. We denote the thus arising finite-state verification tasks as checking whether $S_n \models_{wf} A\ \varphi(1, ..., k)$ holds.

As in Section 2.2.3, we now first state a basic cut-off lemma and then we generalise it. However, the way we establish the cut-off turns out to be significantly more complex because lifting a counterexample behaviour from a small system to a big one is now much more involved than previously. To ensure weak process fairness, newly added processes must be allowed to fire some transitions, but at the same time, this cannot influence the behaviour of the visible processes.

Lemma 2.2.4 *For systems of an RTR\P family $\mathcal{F}$ with m resources and an MPTL path formula $\varphi(p_1, ..., p_k)$, the following holds:*

$$\forall n \geq m+k : S_n \models_{wf} \forall p_1, ..., p_k : A\ \varphi(p_1, ..., p_k) \Leftrightarrow S_{m+k} \models_{wf} A\ \varphi(1, ..., k)$$

Proof. (Sketch) The case of ($\Leftarrow$) is similar to Lemma 2.2.2. The processes that keep running forever in a counterexample behaviour β_n in S_n will run even when we remove some processes. To keep blocked the visible processes that eventually block in β_n, we need at most one auxiliary invisible process per resource.

($\Rightarrow$) We show that we can extend a counterexample behaviour β_{m+k} to one of S_n by allowing the additional processes to fire some steps without influencing the visible behaviour. We distinguish three cases:

Case 1. If all processes block in a counterexample behaviour β_{m+k}, then in S_n, we let them block in the same way with the additional processes idling at q_0. Subsequently, the additional processes can also block by replaying the steps of any of the original processes.

Case 2. If all processes keep running forever in β_{m+k}, and eventually none of them releases any resources any more, at least one process eventually runs without using any resources any more—m resources can be used by at most m processes. The behaviour of such a process can be easily mimicked by the additional processes. (They first go to the location from which they do not use any resources any more, and their behaviour can then be arbitrarily interleaved with the behaviour of the original processes.) Otherwise, at least one resource is always eventually released. As $m-1$ resources can be used by $m-1$ processes only, a configuration in which $k+1$ processes do not use any resource is being regularly passed. At least one of these processes (say p) is invisible. In RTR\P, if p regularly passes a configuration in which it does not use anything, it must regularly pass a configuration in which it does not use nor request anything. The additional processes can then mimic the

behaviour of p such that p plays its original role up to completing one loop on the configuration where it does not use nor request any resource, then the same is done by one of the new processes (which get into the appropriate control location before all other processes start), then another one, and so on up to the last one when the scenario starts repeating. (As the concerned processes do not interfere with any resource in the given state, they do not block each other nor the other processes.)

Case 3. The remaining scenario, in which some processes keep running forever and some block in β_{m+k}, is the most subtle one, but it may be split to several subcases that can be shown to be solvable similarly to Case 1 or Case 2. $\square$

Now, the theorem generalising the lemma can be easily obtained by exploiting properties of MPTL.

Theorem 2.2.5 *Let $\mathcal{F}$ be an RTR\P family with m resources and let $\Phi_a \equiv \forall p_1, ..., p_k : A\ \varphi(p_1, ..., p_k)$ be an MPTL formula. Then, checking whether $\mathcal{F}, l \models^a_{wf} \Phi_a$ holds is equal to checking whether $S_n \models_{wf} A\ \varphi(1, ..., k)$ holds for all $S_n \in \mathcal{F}$ such that $min(max(l, k), m + k) \leq n \leq m + k$.*

Proof. (Sketch) With respect to Lemma 2.2.4, we can use a similar argument as in the proof of Theorem 2.2.1, but as the system size of k (when we can already choose the given number of distinct processes) and the cut-off size of $m + k$ allowing us to generalise the results do not coincide, we have to separately examine each $S_n \in \mathcal{F}$ where $k \leq n \leq m + k$ and $l \leq n$. $\square$

Let us note that examining the systems S_k (if $l \leq k$) and S_{m+k} is necessary for the general result presented in Theorem 2.2.5: Fig. 2.2 (a) shows the RTR\P automaton of a simple family for which $\forall p : A\ \Diamond\Box\ \neg at(p, \mathtt{x})$ does not hold in S_k (i.e., S_1), but it holds in all bigger systems. Fig. 2.2 (b) shows the automaton of a simple family for which $\forall p : A\ \Diamond\ (at(p, \mathtt{x}) \vee at(p, \mathtt{y}))$ does not hold in S_{m+k} (i.e., S_3 where two processes may deadlock and another will not even pass the first request), but it holds in any smaller system. The question of a potential further optimisation of the presented result by not having to examine all the systems between $max(l, k)$ and $m + k$ remains open for the future, but this does not seem to be a real obstacle to practical applicability of the result.

Absence of Structure-Independent Cut-Offs for RTR families

In verification of weakly fair behaviour of RTR families against MPTL formulae, we examine complete, usually infinite behaviours of systems of the considered families. However, to be able to examine such behaviours, we need to examine their finite prefixes as well. Then, Theorem 2.2.2 immediately shows

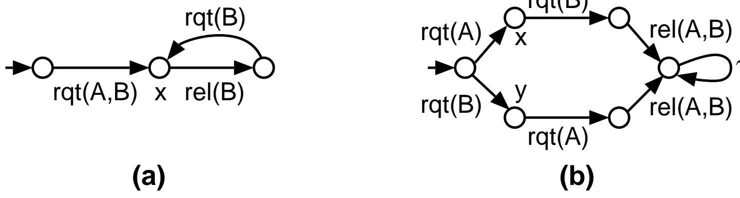


Fig. 2.2. Two simple RTR\P automata

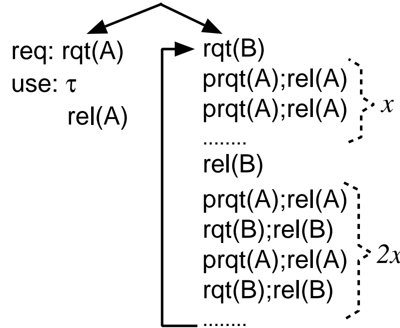


Fig. 2.3. An RTR automaton problematic for verification of absence of starvation

that there does not exist any structure independent cut-off allowing us to reduce the given general problem to finite-state verification. Moreover, for the case of verifying fair behaviour of RTR families against MPTL formulae, no structure-independent cut-offs exist even for more restricted scenarios than in finite behaviour verification. Namely, the query used in the proof of Theorem 2.2.2 speaks about two processes. However, below, we give a theorem showing that for the case of parametric verification of weakly fair behaviour, no structure-independent cut-off exists even for *single-process MPTL formulae*, i.e., formulae having a single process variable and thus speaking about a single visible process. In particular, such a cut-off does not exist for a single-process formula encoding absence of starvation.

Theorem 2.2.6 *For RTR families $\mathcal{F}$ with m resources and the property of absence of starvation expressed as $\Phi_a \equiv \forall p : A \ \Box \ (at(p, req) \Rightarrow \Diamond \ at(p, use))$, the problem of checking whether $\mathcal{F}, 1 \models^a_{wf} \Phi_a$ holds is not, in general, decidable by examining just the systems $S_1, ..., S_n \in \mathcal{F}$ with n being a function of m only.*

Proof. (Sketch) To witness starvation in the RTR automaton in Fig. 2.3, we need at least $x + 3$ processes with x depending on the number of transitions only. The starving process is blocked in `rqt(A)`. The blocking processes run in

the right control branch. When such a process releases `A`, another has to block it instead. A process running in the upper exclusive part of the right branch needs several times a help from the other processes. However, none of the latter processes can help the former several times in a single loop because the former process prevents the latter ones from going from one lower `A`-blocking section to another. □

A Cut-Off for Single-Process MPTL Formulae

There is no simple cut-off for verification of weakly fair behaviours of RTR families against single-process MPTL formulae since a lot of invisible processes requesting resources with a high priority may be needed to block a visible process. Their number depends on the structure of the control automaton. However, this number can be bounded as shown in this section.

To give the bound, we need some definitions. Let $\mathcal{F}(\mathcal{A})$ be an RTR family with m resources. The set of control locations Q_t of $\mathcal{A}$ is split into two disjoint parts: Q_o (all internal control locations and those where processes own at least one resource, without loss of generality a process owns always the same resources at a given control location) and Q_n (the others). Let $F = |Q_n|$ ($F \geq 1$ as Q_n contains the initial location q_0), $C = |2^{Q_o}| = 2^{|Q_o|}$ and $M_C = C^C$. Then, we can define the needed bound as $B_{\mathcal{F}} = CM_C(M_C+1)(2FC(M_C+1))^F + 2C(M_C+1) + 2m + 1$.

The key cut-off lemma below shows that if a formula is true in systems having between $m+1$ and $B_{\mathcal{F}}$ processes, it is also true in systems with more than $B_{\mathcal{F}}$ processes. This and Lemma 2.2.7 stating the opposite allows us to reduce the parametric verification problem to verification of systems with up to $B_{\mathcal{F}}$ processes.

Lemma 2.2.5 *Let $\mathcal{F}$ be an RTR family with m resources and $\varphi(p)$ an MPTL path formula. Then the following holds for systems of $\mathcal{F}$:*

$$\forall n \geq B_{\mathcal{F}} : (\forall n', m{+}1 \leq n' \leq B_{\mathcal{F}} : S_{n'} \models_{wf} \forall p : A\ \varphi(p)) \Rightarrow S_n \models_{wf} \forall p : A\ \varphi(p)$$

Proof. (Sketch) We have to show that given a weakly fair counterexample behaviour β_n in S_n, we can obtain a weakly fair counterexample behaviour $\beta_{n'}$ in $S_{n'}$ with $m{+}1 \leq n' \leq B_{\mathcal{F}}$. There are several cases to consider. The most difficult one is the case where in β_n, the visible process is blocked forever, and invisible processes are running. To make sure that the visible process stays blocked, and the behaviour remains weakly fair, we have to preserve some invisible processes in $\beta_{n'}$. As shown in Section 2.2.4, this number cannot be bounded independently of the structure of the system. We show that less than $B_{\mathcal{F}}$ processes suffice. This is done basically in two steps.

First, we show that we can always reorder transitions fired in the counterexample behaviour β_n in such a way that the contents of queues is restricted

such that it can be deduced from the control location of the involved processes (not considering forever blocked processes).

Second, we show that only at most $B_{\mathcal{F}}$ processes are needed to obtain a behaviour where the visible process stays blocked. To prove this we note that in a behaviour of a system reordered as mentioned above, the number of invisible processes that can be at control locations from Q_o is bounded by one, whereas the number of invisible processes that can be at control locations from Q_n is not bounded. At the same time, the exact identity of invisible processes is not important. Thus, the relevant information of a configuration is which control locations of Q_o are occupied ($\mathsf{q} \in 2^{Q_o}$) and how many processes are at each control location of Q_n ($\boldsymbol{x} \in \mathbb{N}^F$). The number of possible values of q is bounded by C, and so it remains to show that the values of $\boldsymbol{x}$ can be bounded too.

We can suppose that the counterexample behaviour β_n consists of a prefix followed by a loop. From this loop, we can extract the relevant information on configurations. In this way, we obtain a *quotient loop behaviour* which is a sequence of couples $(\mathsf{q}, \boldsymbol{x})$. Lemma 2.2.6 given below then provides us with a smaller quotient loop behaviour using less than a bounded number of processes and satisfying some additional conditions. This quotient loop can be used to obtain a weakly fair behaviour using less than $B_{\mathcal{F}}$ (and at least $m + 1$) processes. □

Lemma 2.2.6 *If there is a quotient loop $\overline{\gamma}$ of length bigger than $S = CM_C(M_C + 1)(2FC(M_C + 1))^F + 2C(M_C + 1)$ starting from a configuration $(\mathsf{q}, \boldsymbol{x})$, then there is a quotient loop $\overline{\gamma'}$ of length smaller than or equal to S starting from a configuration $(\mathsf{q}, \boldsymbol{x}')$. Furthermore, (1) for all configurations $(\mathsf{p}, \boldsymbol{y})$ in $\overline{\gamma'}$, we have $|\mathsf{p}| + \sum_{i=1}^{F} \boldsymbol{y}_i \leq S$, (2) a transition appearing in $\overline{\gamma}$ appears at least once in $\overline{\gamma'}$, (3) $\forall i$ with $1 \leq i \leq F$, there exists a configuration $(\mathsf{p}, \boldsymbol{y})$ in $\overline{\gamma'}$ with $\boldsymbol{y}_i = 0$.*

Proof. (Sketch) $\overline{\gamma}$ can be decomposed into repeating short *structural loops* (i.e., quotient behaviours starting from some configuration $(\mathsf{p}, \boldsymbol{y})$ and going to $(\mathsf{p}, \boldsymbol{y'})$ where the occupied locations in Q_o are the same, but the number of invisible processes in locations Q_n may change) and some short inner quotient behaviours not containing such loops. Then, we construct a linear equation of the form $\mathbf{A}\boldsymbol{x} = \mathbf{B}$ where, intuitively, $\mathbf{A}$ encodes the effect of all short structural loops on vectors $\boldsymbol{y} \in \mathbb{N}^F$ and $\mathbf{B}$ encodes the effects of short inner quotient behaviours. The entries of both $\mathbf{A}$ and $\mathbf{B}$ are bounded. We know that there is a solution $\boldsymbol{x}$ for this equation given by $\overline{\gamma}$ which indicates how many times the short loops have to be repeated. Then, we use a lemma from the theory of Linear Integer Programming [vZGS78] showing that if $\mathbf{A}\boldsymbol{x} = \mathbf{B}$ has a solution and entries of $\mathbf{A}$ and $\mathbf{B}$ are bounded, then it has a bounded solution from which we construct a loop behaviour of the required size satisfying the three conditions of the lemma. □

We now formalise the counterpart to Lemma 2.2.5. We show that if the weakly fair behaviours of a system with more than $m+1$ processes satisfy some single-process MPTL formula (it suffices when this holds for process 1 being visible), then the formula is satisfied for the smaller systems (with at least $m+1$ processes) too. Subsequently, we use this fact together with the above lemmas to give a complete cut-off result for single-process MPTL formulae and weakly fair behaviour of systems of RTR families.

Lemma 2.2.7 *Let $\mathcal{F}$ be an RTR family and $\varphi(p)$ an MPTL path formula. Then, for systems of $\mathcal{F}$, we have: $\forall n' \geq m+1, n \geq n' : S_n \models_{wf} A\ \varphi(1) \Rightarrow S_{n'} \models_{wf} \forall p\!: A\ \varphi(p)$*

Proof. A straightforward analogy of the proof of ($\Rightarrow$) in Lemma 2.2.4. □

Theorem 2.2.7 *Let $\mathcal{F}$ be an RTR family with m resources and let $\Phi_a \equiv \forall p : A\ \varphi(p)$ be a single-process MPTL formula. Then, checking $\mathcal{F}, l \models^a_{wf} \Phi_a$ is equal to checking $S_n \models_{wf} A\ \varphi(1)$ for all $S_n \in \mathcal{F}$ with $l \leq n \leq m+1$ or $n = B_{\mathcal{F}}$.*

Proof. If there are $S_n \in \mathcal{F}$ with $l \leq n \leq m+1$, they are covered directly exploiting just the interchangeability of processes. Then, for the ($\Leftarrow$) case, if $S_{B_{\mathcal{F}}} \models_{wf} A\ \varphi(1)$ holds, Lemma 2.2.7 implies $S_n \models_{wf} \forall p : A\ \varphi(p)$ holds for $m+1 \leq n \leq B_{\mathcal{F}}$. Using Lemma 2.2.5, this can be extended to hold for $S_n \in \mathcal{F}$ with $m+1 \leq n$, which is what is needed or even more (for $m+1 < l$). For the ($\Rightarrow$) case, If $S_n \models_{wf} \forall p : A\ \varphi(p)$ holds for $l \leq n$, $S_{B_{\mathcal{F}}} \models_{wf} A\ \varphi(1)$ follows either directly for $l \leq B_{\mathcal{F}}$ or from Lemma 2.2.7 (note that $m+1 \leq B_{\mathcal{F}}$). □

Simple RTR Families

Above, we have shown that parametric verification of weakly fair behaviour of RTR families against single-process MPTL formulae is decidable, but no really simple reduction to finite-state verification is possible in general. We now give a restricted (yet still meaningful) subclass of RTR families for which the problem is simpler and can be solved using a structure-independent cut-off bound.

An RTR family $\mathcal{F}$ is *simple* if the set of control locations Q_n contains only the initial location q_0: Processes start from it by requesting some resources (possibly in different ways) and then they may request further resources as well as release some resources. However, as soon as they release all of their resources, they go back to q_0. This class is not unrealistic; it corresponds to systems with a single resource-independent computational part surrounded by actions using resources.

For simple RTR families, we show an improved cut-off bound using $2m+2$ processes, which is better than $B_{\mathcal{F}}$ for $F=1$. This is basically due to the fact that only m invisible processes can be simultaneously in control locations Q_o.

Lemma 2.2.8 *Let $\mathcal{F}$ be a simple RTR family with m resources and $\varphi(p)$ an MPTL path formula. Then, the following holds for systems of $\mathcal{F}$:*

$$\forall n \geq 2m+2 : \\ (\forall n', m+1 \leq n' \leq 2m+2 : S_{n'} \models_{wf} \forall p : A\ \varphi(p)) \Rightarrow S_n \models_{wf} \forall p : A\ \varphi(p)$$

Proof. (Sketch) We only show here the case where the visible process is blocked and some invisible processes are running. We proceed as in the proof of Lemma 2.2.5 to get a counterexample behaviour β_n with bounded queues. Since $|Q_n| = 1$, the relevant information of a configuration is a couple (p, x) with $\mathsf{p} \in 2^{Q_o}$ and $x \in \mathbb{N}$. From β_n (without forever blocked processes from which we have to preserve at most m invisible ones), we can obtain a quotient behaviour where for all pairs (p, x), we have $|\mathsf{p}| + x \leq m+1$. Therefore, we need to preserve at most $m+1$ running processes. Together with the visible and the m blocked invisible ones, this gives at most $2m+2$. □

Theorem 2.2.8 *Let $\mathcal{F}$ be a simple RTR family with m resources and let $\Phi_a \equiv \forall p : A\ \varphi(p)$ be a single-process MPTL formula. Then, checking whether $\mathcal{F}, l \models^a_{wf} \Phi_a$ holds is equal to checking whether $S_n \models_{wf} A\ \varphi(1)$ holds for all $S_n \in \mathcal{F}$ such that $l \leq n \leq m+1$ or $n = 2m+2$.*

Proof. As the proof of Theorem 2.2.7 using Lemma 2.2.8 instead of Lemma 2.2.5. □

Notice that an invisible process can freely move among all locations in a subcomponent Q' of $\mathcal{A}$ which is strongly connected by τ-transitions. Therefore, Theorem 2.2.8 can be generalised to families whose Q_n corresponds to such a component. Moreover, the same idea can be used to optimise the general $B_{\mathcal{F}}$ bound.

2.2.5 Process Deadlockability

Given an RTR family $\mathcal{F}$ and a system $S_n \in \mathcal{F}$, we say that a *process p is deadlocked* in a configuration $c \in C_n$ if there is no configuration reachable from c from which we could fire some transition in p. As is common for linear-time frameworks, process deadlockability cannot be expressed in MPTL, and so since it is an important property to check for the class of systems we consider, we now provide a specialised (structure-independent) cut-off result for dealing with it.

Theorem 2.2.9 *Let $\mathcal{F}$ be an RTR family with m resources. For any l, the systems $S_n \in \mathcal{F}$ with $l \leq n$ are free of process deadlock iff $S_{max(m,2)} \in \mathcal{F}$ is.*

Proof. (Sketch) A full proof of this fact is quite subtle. We can encounter scenarios where a group of processes is mutually deadlocked due to some circular dependencies in queues of requests, but also situations where a process is deadlocked due to being always inevitably overtaken by processes that keep running and do not even own any resource forever. However, when we (partially) replace overtaking by postponed firing of requests (cf. Lemma 2.2.3), when we push blocked high-priority requests before the low-priority ones (the former block the latter, but not vice versa), and when we preserve only the running processes that never release all resources at the same time, we can show that we suffice with one (primary) blocked and/or blocking process per resource. □

Let us note that the possibility of inevitable overtaking examined in the proof of Theorem 2.2.9 as a possible source of process deadlocks in systems of RTR families is stronger than starvation. Starvation arises already when there is a single behaviour in which some process is eventually always being overtaken. Interestingly, as we have shown, inevitable overtaking is much easier to handle than starvation, and we obtain a cut-off bound that cannot be improved even when we restrict ourselves to RTR\P families with no overtaking.

2.2.6 RTR Families and Undecidability

Finally, we discuss an extension of MPTL by *local process quantification* [ES97] where processes to be monitored in a behaviour are not fixed at the beginning, but may be chosen independently in each encountered state. Local process quantification can be added to MPTL by allowing $\forall V' : \varphi(V \cup V')$ to be used in a path formula $\varphi(V)$ with the semantics $\pi_n, \nu_n \models \forall V' : \varphi(V \cup V')$ iff $\pi_n, \nu'_n \models \varphi(V \cup V')$ holds for all valuations ν'_n of PV such that $\forall p \in PV \setminus V' : \nu'_n(p) = \nu_n(p)$. Such a quantification can be used to express, e.g., the global response property $A\Box((\exists p_1 : at(p_1, req)) \Rightarrow \Diamond(\exists p_2 : at(p_2, resp)))$, which cannot be encoded with global process quantifiers if the number of processes is not known. Unfortunately, it can be shown that parametric verification of linear-time finite-behaviour properties with local process quantification is undecidable even for RTR\P families.

Theorem 2.2.10 *The parameterised finite-behaviour verification problem of checking whether $\mathcal{F}, 1 \models^a_{fin} \Phi_a$ holds for an RTR\P family $\mathcal{F}$ and an MPTL formula Φ_a with local process quantification is undecidable even when the only temporal operators used are $\Box$ and $\Diamond$ and no temporal operator is in the scope of any local process quantifier.*

Proof. (Idea) The result can be proved by a reduction from the problem of checking nonemptiness of languages of push-down automata (PDAs) with two push-down stacks. Simulation of PDAs by RTR\P families is quite complex due to the following facts: The queues in RTR are used as waiting queues, and not communication queues. They contain just the identities of waiting processes, all the processes have identical control, no process can manipulate queue items of other processes, and a process cannot even know whether or not it will have to wait when requesting some resource (nor whether it had to wait for a resource it is using now).

We only present the main ideas of the construction here. We simulate stack symbols as well as control states of a given two-stack PDA M by processes running in different branches of a suitably designed RTR\P automaton. (Input symbols need not be taken into account.) For each occurrence of a state or a stack symbol in a run of M, a fresh process is used. The processes wait for their use in the queue of a certain resource, and parameterisation assures that there are enough of them. The stacks of M are simulated by queues of requests. The simulation is driven by processes corresponding to encountered control states of M. Such processes always control most of the resources used in the construction, and thus keep other processes (representing the current content of the stacks or just waiting for their future use) in the appropriate queues.

By blocking some resources and unblocking others, a current-state process can allow a certain process to leave the queue simulating some PDA stack (which corresponds to the pop operation), some processes to enter such a queue (simulating push), and some new state process to take over its role (simulating a passage to a new control state of M). The simulation is such that if a process not representing the right symbol or not being at the top leaves a stack, the whole simulation deadlocks without reaching an acceptance state. (For example, to check which symbol a process represents, we have a special resource for each symbol, only one of these resources is unblocked, and the appropriate process has to pass through the resource corresponding to the symbol it simulates.) Similar constructions can be used to implement the other needed operations.

The above construction guarantees that when the involved processes do not cause a premature global deadlock nor ignore any opportunity to exhibit some progress, their activities correspond to a correct simulation of M. A violation of the latter condition cannot be detected and handled in an RTR\P automaton, but we can use a suitable MPTL formula with local process quantification to focus on runs where the processes really exhibit the activities they are supposed to exhibit. (For example, we check whether if some process is supposed to leave a stack, some process really leaves it, etc.) Finally, correctness of the initialisation of the simulation and the fact that an accepting state is reached can be checked via MPTL with local process quantification too. □

2.3 A Summary on Cut-offs and RTR Families

We have presented the approach of finding cut-offs that may allow us to transform some infinite-state/parameterised verification problems to (a series of) finite-state ones. If this step is successful and leads to reasonable bounds on the involved sources of infinity and/or parameterisation, it may yield a very efficient verification procedure exploiting the already quite elaborated finite-state model checkers.

We have briefly overviewed a number of cut-off results on different kinds of systems: networks of processes with a single control process and many identical user processes, networks of processes communicating by token passing, networks of processes with disjunctive or conjunctive guards, process networks with resource sharing, cache coherence protocols, and systems with parameterised-size arrays.

Then, we have in detail discussed our original results on verification of parameterised process networks with resource sharing. We have defined an abstract model for a significant class of parametric systems of processes competing for access to shared resources under a FIFO resource management with a possibility of distinguishing low- and high-priority requests. The primitives capturing the interaction between processes and resources and the resource management policies considered are natural and inspired by real-life applications. We have established cut-off bounds showing that many practical parametric verification problems (including verification of mutual exclusion, absence of starvation, and process deadlockability) are decidable in this context. The way the obtained results were established is sometimes technically highly involved, which is due to the fact that the considered model is quite powerful and (as we have also shown) positive decidability can easily be lost if verification of a bit more complex properties is considered.

The structure-independent cut-offs we have presented are small and—for verification of finite behaviour and process deadlockability—optimal. They provide us with practical decision procedures for the concerned parametric verification problems and, moreover, they can also be used to simplify finite-state verification for systems with a given, large number of processes.

The structure-dependent cut-off for single-process formulae in the case of verifying the fair behaviour of the general RTR families is quite big and does not yield a really practical decision procedure. One challenging problem interesting for the future is to optimise this bound. Although we know that no general structure-independent cut-off exists, the bound we have provided is not optimal, and significantly improved cut-offs could be found especially for particular classes of systems as we have already shown for simple RTR families.

Another interesting problem is to improve the decidability bounds. For general RTR families and arbitrary MPTL formulae, decidability of parametric verification of finite as well as fair behaviour is still open. So far, we have only shown that these problems cannot be handled via structure-independent

cut-offs. Conversely, the question of existence of practically interesting, decidable fragments of MPTL with local process quantification is worth examining too. For the cases where no (or no small) cut-off can be found, we could then try to find some adequate abstraction techniques and/or symbolic verification techniques.

Finally, several extensions or variants of the framework can be considered. For example, the questions of non-exclusive access to resources or nonblocking requests can be examined. Moreover, several other locker policies can be considered, e.g., service in random order or a policy where any blocked process can be overtaken. We believe that the results presented here and the reasoning used to establish them provide (to a certain degree) a basis for examining such questions.

3

Regular Model Checking

In the previous chapter, we have discussed an approach to formal verification of infinite-state systems based on reducing their verification to (series of) finite-state verification problems that can be handled using traditional model checking techniques. If such a step is not possible (or not efficient), we can instead use symbolic model checking techniques based on a suitable finite representation of infinite sets of reachable configurations. For this purpose, one can exploit, e.g., various kinds of logic (in the literature, one can find, for instance, approaches based on WS1S, Presburger arithmetics, predicate logic with transitive closure, etc.). Another widely used possibility—which we concentrate here on—is to deal with *regular sets of configurations encoded by finite-state automata* (as we will see later, omega-regular and tree-regular sets are also often in use).

Regular sets have proved useful for representing sets of reachable configurations of many different kinds of systems including systems with unbounded queues (communication channels), push-down stacks (recursion), counters, parameterised numbers of components, etc. Introducing a finite representation of infinite sets is, however, not the only problem to be faced in infinite-state model checking. Further, one has to represent the set of transitions of an infinite-state system in a finite way too. A good point for the use of regular languages as a symbolic encoding is that often the set of transitions forms a *regular relation* that may be represented, for instance, using a finite-state transducer (which is a natural counterpart to using finite-state automata for representing sets of configurations).

Another issue is then how to explore *firing sequences* of the transitions. Clearly, we cannot explore the firing sequences by firing the infinitely many transitions one-by-one. Moreover, even if the number of the transitions is finite, an infinite non-looping firing sequence can be obtained in an infinite-state system—consider, e.g., a simple loop adding some element into a push-down stack. Consequently, one has to come up with a way of exploring infinite numbers of transitions of the considered systems at once.

The methods proposed for computing the effect of possibly infinite numbers of transitions over regular sets of configurations may be divided into domain-specific and generic ones. *Domain-specific methods* have been proposed, for instance, for dealing with unbounded FIFO communication channels [BP96, BGWW97, WB98], lossy FIFO channels [ABJ98, AJ96a, CFI96a, ABB01], integers [BP94, WB95, WB98, FL02, BB04], and push-down systems [BEM97, FWW97, EHRS00]. *Generic techniques* appeared originally in the world of parameterised networks of processes [KMM$^+$97, KMM$^+$01, ABJN99, BJNT00], but they can be applied in all the mentioned areas and even in some other ones (as, e.g., in the verification of programs with dynamic linked data structures as described in [BHMV05] and in Chapter 4).

In this work, we in particular concentrate on the generic approach denoted usually as *regular model checking*[1] (RMC). Regular model checking is based on dealing with regular sets of reachable configurations encoded by finite-state automata and regular transition relations represented by finite-state transducers. It uses iterative applications or compositions of the finite-state transducers accelerated in a suitable way in order to compute in a finite time the effect of an infinite number of transitions of the system being examined, and thus make the analysis terminate as often as possible. Note that in general, termination cannot be guaranteed as the problems being solved by regular model checking are mostly undecidable, and, indeed, it is easy to encode the one-step transition relation of a Turing machine using a finite-state transducer.

Below, we first define the basic notions of the automata theory that we need, then we explain the basic idea of regular model checking, the way it can address various verification problems, and we provide an overview of the various acceleration approaches proposed for making regular model checking terminate as often as possible. Then, we concentrate on *abstract regular model checking* and *regular model checking based on language inference*, where our original contribution—achieved in a tight cooperation with our partners—is situated. Finally, at the end of the section, we mention several extensions of the basic regular model checking framework.

3.1 Finite-State Automata and Transducers

A (non-deterministic) *finite-state automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ a transition function, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of final states. The transition relation $\underset{M}{\longrightarrow} \subseteq Q \times \Sigma^* \times Q$ of M is defined as the smallest relation satisfying:

[1] Note that the term "regular model checking" is sometimes used to cover both the generic and domain-specific methods as long as they work with regular sets of states and transitions having the form of a regular relation. Generic and domain-specific regular model checking can then distinguished. In this work, we identify regular model checking with generic regular model checking as also often done in the literature.

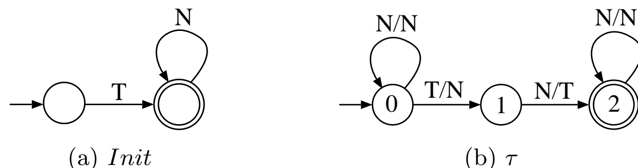
(1) $\forall q \in Q : q \xrightarrow[M]{\varepsilon} q$, (2) if $q' \in \delta(q, a)$, then $q \xrightarrow[M]{a} q'$, and (3) if $q \xrightarrow[M]{w} q'$ and $q' \xrightarrow[M]{a} q''$, then $q \xrightarrow[M]{wa} q''$ for $a \in \Sigma, w \in \Sigma^*$. The subscript M may be dropped if no confusion is possible. The automaton is called *deterministic* iff $\forall q \in Q\ \forall a\ \in \Sigma : |\delta(q, a)| \leq 1$.

The *language* recognised by a finite-state automaton $M = (Q, \Sigma, \delta, q_0, F)$ from a state $q \in Q$ is defined by $L(M, q) = \{w \in \Sigma^* \mid \exists q_F \in F : q \xrightarrow[M]{w} q_F\}$. The language $L(M)$ of M is equal to $L(M, q_0)$. A set $L \subseteq \Sigma^*$ is a *regular set* iff there exists a finite-state automaton M such that $L = L(M)$. We also define the *backward language* $\overleftarrow{L}(M, q) = \{w \mid q_0 \xrightarrow[M]{w} q\}$ and the *forward/backward languages of words up to a certain length*: $L^{\leq n}(M, q) = \{w \in L(M, q) \mid |w| \leq n\}$ and similarly $\overleftarrow{L}^{\leq n}(M, q)$. We define the *forward/backward trace languages* of states $T(M, q) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^* : ww' \in L(M, q)\}$ and similarly $\overleftarrow{T}(M, q)$. Finally, we define accordingly forward/backward trace languages $T^{\leq n}(M, q)$ and $\overleftarrow{T}^{\leq n}(M, q)$ of *traces up to a certain length*.

Given a finite-state automaton $M = (Q, \Sigma, \delta, q_0, F)$ and an equivalence relation $\sim$ on its set of states Q, $M_{/\sim}$ denotes the *quotient automaton* of M wrt. $\sim$, $M_{/\sim} = (Q_{/\sim}, \Sigma, \delta_{/\sim}, [q_0]_{/\sim}, F_{/\sim})$ where $Q_{/\sim}$ and $F_{/\sim}$ are the partitions of Q and F wrt. $\sim$, respectively, $[q_0]_{/\sim}$ is the equivalence class of Q wrt. $\sim$ containing q_0, and $\delta_{/\sim}$ is defined such that $[q_1]_{/\sim} \xrightarrow[M_{/\sim}]{a} [q_2]_{/\sim}$ iff $q_1 \xrightarrow[M]{a} q_2$ for $[q_1]_{/\sim}, [q_2]_{/\sim} \in Q_{/\sim}$, $a \in \Sigma$. A *finite-state transducer* over Σ is a 5-tuple $\tau = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite input/output alphabet, $\delta : Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon \to 2^Q$ a transition function, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of final states. A finite-state transducer is called a *length-preserving transducer* if its transitions do not contain ε. The transition relation $\xrightarrow[\tau]{} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is defined as the smallest relation satisfying: (1) $q \xrightarrow[\tau]{\varepsilon/\varepsilon} q$ for every $q \in Q$, (2) if $q' \in \delta(q, a, b)$, then $q \xrightarrow[\tau]{a/b} q'$, and (3) if $q \xrightarrow[\tau]{w/u} q'$ and $q' \xrightarrow[\tau]{a/b} q''$, then $q \xrightarrow[\tau]{wa/ub} q''$ for $a, b \in \Sigma_\varepsilon, w, u \in \Sigma^*$. The subscript τ may again be dropped if no confusion is possible. A finite-state transducer $\tau = (Q, \Sigma, \delta, q_0, F)$ represents the *relation* $\varrho(\tau) = \{(w, u) \in \Sigma^* \times \Sigma^* \mid \exists q_F \in F : q_0 \xrightarrow[\tau]{w/u} q_F\}$. A relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is a *regular relation* iff there exists a finite-state transducer τ such that $\varrho = \varrho(\tau)$. For a set $L \subseteq \Sigma^*$ and a relation $\varrho \subseteq \Sigma^* \times \Sigma^*$, we denote $\varrho(L)$ the set $\{w \in \Sigma^* \mid \exists w' \in L : (w', w) \in \varrho\}$.

3.2 Regular Model Checking: The Basic Idea

The basic idea behind regular model checking is to encode particular configurations of the considered systems as *words over a suitable finite alphabet*



(a) *Init* (b) τ

Fig. 3.1. A model of a simple token passing protocol: (a) an automaton *Init* modelling the initial set of configurations $I = L(Init)$, and (b) a transducer τ modelling the one-step transition relation $\varrho = \varrho(\tau)$

and to represent infinite, but regular sets of such configurations by *finite-state automata*. Regular transition relations between the configurations are then encoded using *finite-state transducers*.

For example, when dealing with parameterised networks of finite-state processes, each letter in a word will typically model the state of a single process, and the length of the word will correspond to the number of processes in the given instance of the system. To illustrate the idea, let us consider a very simple token passing protocol. We have an arbitrary, but finite number of processes arranged into a linear network. Each process either does not have a token and is waiting for a token to arrive from its left neighbour, or it has a token and then it can pass it to its right neighbour. We suppose that initially there is only one token which is owned by the left-most process. To encode the state of each process in our protocol, we suffice with the alphabet $\Sigma = \{N, T\}$ where N means that the process does not have a token whereas T means the process has a token. Then, the set I of all possible initial configurations may be encoded by the automaton *Init* shown in Fig. 3.1(a) and the single-step transition relation by the transducer τ in Fig. 3.1(b).

As for applications of regular model checking in other areas, encoding of push-down stacks and queues as words is straightforward. Further, sets of vectors of integers may be represented, e.g., in the form of the so-called *number decision diagrams* (NDDs) [WB95] where the integers are encoded in binary—in general, a different base $r > 1$ can also be used—and put in parallel (i.e., in a word, we have a sequence of values of the 0th-order bits of all the elements of the vector, then a sequence of all the 1st-order bits, the 2nd-order bits, and so on) with negative numbers expressed using the 2's complement.[2] In Chapter 4, we will then show how to use words and automata to encode even more complex configurations, namely configurations of programs with dynamic linked data structures with one selector (lists, circular lists).

[2] Note that if we wrote the particular members of a vector in a series, we would not even be able to capture in a regular way that they should be the same while NDDs have a strictly greater expressive power than the Presburger arithmetics.

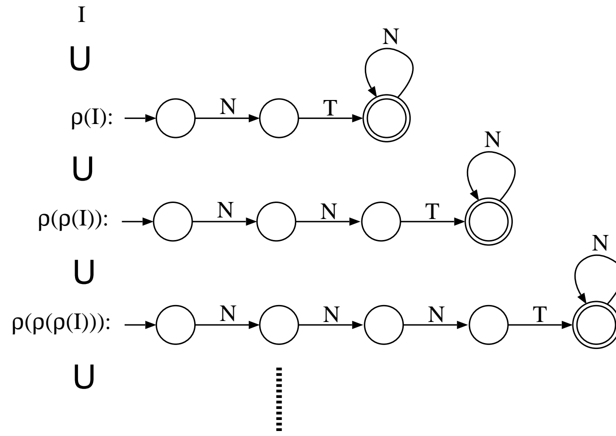


Fig. 3.2. Divergence of the non-accelerated reachability set computation for the simple token passing protocol $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup ...$

Let us, however, get back to the main principles of regular model checking. Once we have a transducer encoding the single-step transition relation ϱ of the system that we want to examine and an automaton encoding its set of initial configurations I, there are two basic strategies we can follow. We can either try to directly compute the set of all reachable configurations $\varrho^*(I)$, or the reachability relation ϱ^* of the system. The set $\varrho^*(I)$ can be obtained by repeatedly applying the single-step transition relation ϱ on the set of the so-far reached states and by taking the union of all such sets, i.e., $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup ....$ On the other hand, the reachability relation ϱ^* can be obtained by repeatedly composing ϱ with the so-far computed reachability relation and by taking the union of all such relations, i.e., $\varrho^* = \iota \cup \varrho \cup (\varrho \circ \varrho) \cup (\varrho \circ \varrho \circ \varrho) \cup ...$ where ι is the identity relation.

The problem is that in the context of parameterised and infinite-state systems (as opposed to finite-state systems), if we try to compute the above infinite unions using a straightforward fixpoint computation, i.e., if we keep computing and uniting more and more elements of the described sequences till a fixpoint is reached, the computation will usually not terminate. We can illustrate this even on our simple token passing protocol. In Fig. 3.2, we give the first members of the sequence I, $\varrho(I)$, $\varrho(\varrho(I))$, $\varrho(\varrho(\varrho(I)))$, ..., which clearly show that a fixpoint will never be reached (the token can be at the beginning, one step to the right, two steps to the right, three steps to the right, etc.).

In order to make the computation of $\varrho^*(I)$ or ϱ^* terminate at least in many practical cases, we need some kind of *acceleration* of the computation which will allow us to obtain the result of an infinite number of the described computation steps at once (i.e., to in some sense "jump" to the fixpoint). We can, e.g., notice that in our example, the token is moving step-by-step to the

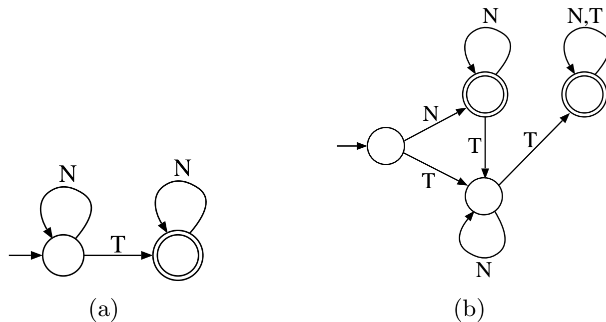


Fig. 3.3. The simple token passing protocol—automata encoding: (a) the set of reachable configurations, and (b) the set of bad configurations

right, and we can accelerate the fixpoint computation by allowing the token to move arbitrarily far to the right in one step. If we use such an acceleration, we will immediately reach the fixpoint shown in Fig. 3.3(a) which represents the set of all reachable configurations of our protocol. In the literature, several different approaches to a systematic acceleration of fixpoint computations in regular model checking have been proposed. We will review them in Section 3.4.

Before proceeding further on to the use of regular model checking for actually verifying some properties of the studied systems, we add a note on computing either $\varrho^*(I)$ or ϱ^*. The ability to compute ϱ^* may be suitable for some methods of checking certain properties of the studied systems (especially in the case of liveness) as we will see in the next section. However, our experimental evidence shows that computing ϱ^* is usually more difficult, and it is in some sense more difficult even from a theoretical point of view. The reason is that there are systems whose reachability relation ϱ^* is not regular despite both ϱ and $\varrho^*(I)$ are regular. As a simple example, imagine the single-step transition relation over the alphabet $\Sigma = \{a, b\}$ that can move an arbitrary long subword consisting of symbols b by one position to the left or right when it is surrounded solely by symbols a (we will use similar relations when handling programs with dynamic linked data structures with one selector in Chapter 4). To express ϱ^*, we need an ability of unbounded counting to remember the length of the moving subword. However, when we apply ϱ^* to a regular I, $\varrho^*(I)$ is regular.

3.3 Verification by Regular Model Checking

In the previous section, we have shown how we can compute the set of reachable configurations or the reachability relation in the framework of regular

model checking. We now discuss how this can be used for checking (linear time) safety and liveness properties of the examined systems.

It is well known that checking of *safety properties* can be reduced to checking that no "bad" states are reachable in the given system—or, in more complex cases, in its product with some safety monitor. Computing such a product is not difficult provided that the safety monitor has the usual form of a finite-state automaton. One additional letter in each configuration word may then encode the current control state of the safety monitor, and the transition relation of the monitor can be combined with the one-step transition relation of the system. We may even add multiple safety monitors—even one for every letter in a configuration word, which may be useful when checking safety on every process in a parameterised network of processes. Then, if the set of bad states, for which we want to check that they are not reachable in the given system, can be expressed as a regular set B, we may simply compute the reachability set $\varrho^*(I)$ and check that $\varrho^*(I) \cap B = \emptyset$.

For instance, in our simple token passing protocol, we can consider as bad the situation when there is no token in the system or when there appear two or more tokens. The set of such bad states is encoded by the finite-state automaton in Fig. 3.3(b), and it is clearly visible that its intersection with the set of reachable states in Fig. 3.3(a) is empty, and thus the system is safe in the given sense.

Checking of *liveness properties* within regular model checking is considerably more difficult. In the world of finite-state systems, it is known that liveness can be reduced to the repeated reachability problem (on the product of the examined system and a Büchi automaton[3] corresponding to the liveness property to be checked). A similar approach can be taken in the context of regular model checking when the studied systems are modelled by *length-preserving transition relations*, which is typical, e.g., for parameterised networks of processes. In such cases, clearly, the only way how a system can loop is to repeatedly go through some configuration. In a similar way as above, we can then instrument the system by the Büchi automaton (or automata) encoding the undesirable behaviours, and check, e.g., that $\varrho^*(I) \cap A \cap domain(\varrho^+ \cap \iota) = \emptyset$. Here, A is the set of accepting configurations, ι is the identity relation, and *domain* is the projection of a relation onto its domain.

Note that in the above described computation, we need to compute not only the reachability set, but also the reachability relation. However, this step may be avoided by guessing when an accepting cycle begins, doubling every letter in the given configuration word, then continuing the computation only on the even letters and detecting a closure of the loop by looking for a situation when all the even letters correspond to the odd ones—we have practically

[3] Büchi automata are finite automata that accept infinite words by infinitely looping through some of their accepting states (for a formal definition and the associated theory see, e.g., [PP03]).

tested this technique in some of the experiments presented in Section 3.5.5 (and it was studied more deeply in [SB05]).

A systematic framework for modelling parameterised networks of processes as well as specifying their properties to be checked via regular model checking has been proposed in [AJN$^+$04]. The framework uses as a modelling as well as a specification language LTL(MSO) that is a combination of the linear time temporal logic LTL for expressing temporal relations and the monadic second-order logic on words for expressing properties on configuration words. (The MSO part is used for specifying, e.g., that every process in a configuration has to satisfy some condition, or that in the configuration there must exist a process for which some condition holds, and so on.) The work also proposes an automatic translation of the models as well as properties to be checked over them into an automata framework suitable for regular model checking.

Finally, checking liveness properties for systems modelled using *non-length-preserving transition relations* is even more complex than checking liveness in the length-preserving case. This is because a non-length-preserving system may exhibit infinite behaviours infinitely going through an accepting state of the monitoring Büchi automaton even when it does not loop at all—it suffices to imagine a system with a queue that keeps growing beyond every bound. For such cases, [BLW05] has proposed an approach based on using regular model checking for automatically computing the greatest simulation relation on the reachable configurations which is compatible with the property being tracked. Then, instead of checking that an accepting configuration can be reached that is reachable from itself too, one checks that an accepting configuration c_1 is reachable from which an accepting configuration c_2 simulating c_1 (i.e., allowing at least the same behaviours from the point of view of the tracked property) is reachable. An alternative approach based on learning fixpoints of specially proposed modalities from their generated samples using language inference algorithms has then been proposed in [VSVA05].

3.4 Acceleration in Regular Model Checking

The methods of acceleration in regular model checking can be divided into several groups—namely, techniques based on acceleration schemes, quotienting, extrapolation, abstraction, and language inference (though this last technique is in principle a bit further from the previous ones). Below, we briefly discuss the first three techniques, and then we describe the last two in more detail in Sections 3.5 and 3.6.

3.4.1 Acceleration Schemes

The use of acceleration schemes has been proposed in [PS00]. Acceleration schemes allow one to derive (from the original transitions of a system) meta-transitions encoding the effect of firing some of the original transitions an

arbitrary number of times. The work [PS00] has provided three particular schemes for which it is experimentally checked that they suffice for verification of many cases of parameterised networks of processes. In particular, the following schemes are considered: (1) *local acceleration* allowing an arbitrary number of successive transitions of a single process to be fired at once, (2) *global acceleration of unary transitions* allowing any number of processes to fire a certain transition in a sequential order within one accelerated step, and (3) *global acceleration of binary transitions* allowing any number of processes to fire in a sequential order two consecutive transitions each—and thus communicate with both of its neighbours—in one atomic step (this way, e.g., a token in a token passing protocol can "jump" any number of positions ahead in one accelerated step). This method has been implemented in the TLV[P] tool [Sha01].

3.4.2 Quotienting

The quotienting technique has been elaborated in the series of works [BJNT00, JN00, Nil00, DLS01, AdJN02, AdJN03, Nil05] and implemented in the Uppsala regular model checking tool [URM].

Let $\tau = (Q, \Sigma, \delta, q_0, F)$ be a length-preserving transducer encoding the single-step transition relation ϱ of a system being examined. The basic idea of the quotienting technique stems from viewing the result of an arbitrary number of compositions of ϱ encoded by τ as an infinite-state "history" transducer $\tau_{hist} = (Q^+, \Sigma, \delta_{hist}, \{q_0\}^+, F^+)$ whose states[4] reflect the history of their creation in terms of which states of τ have been passed at a particular location in a word in the first, second, and further transductions. Therefore, δ_{hist} is defined such that $q_1q_2...q_n \xrightarrow[\tau_{hist}]{a/a'} q'_1q'_2...q'_n$ for some $n \geq 1$ iff there exist $a_1, a_2, ..., a_{n+1} \in \Sigma$ such that $a = a_1$, $a' = a_{n+1}$, and $\forall i \in \{1, ..., n\} : q_i \xrightarrow[\tau]{a_i/a_{i+1}} q'_i$. Intuitively, this means that $q_1q_2...q_n \xrightarrow[\tau_{hist}]{a/a'} q'_1q'_2...q'_n$ represents the composition of the $q_i \xrightarrow[\tau]{a_i/a_{i+1}} q'_i$ transductions for $i = 1, ..., n$. Clearly, τ_{hist} encodes the reachability relation ϱ^+. An example of the history transducer for the token passing example from Section 3.2 is shown in Figure 3.4.

Of course, the history transducer τ_{hist} is of no practical use as it is infinite-state. The idea is to come up with the so-called *column equivalence* $\simeq$ on its states—i.e., on sequences (or, in the original terminology, columns) of states of the original transducer τ—such that the *quotient transducer* $\tau_{hist/\simeq}$ is (1) finite-state as often as possible, and at the same time, (2) describes exactly the same relation as τ_{hist}. In [AdJN03, Nil05], a systematic framework for designing column equivalences is proposed based on *forward* and *backward*

[4] We allow here a set of initial states.

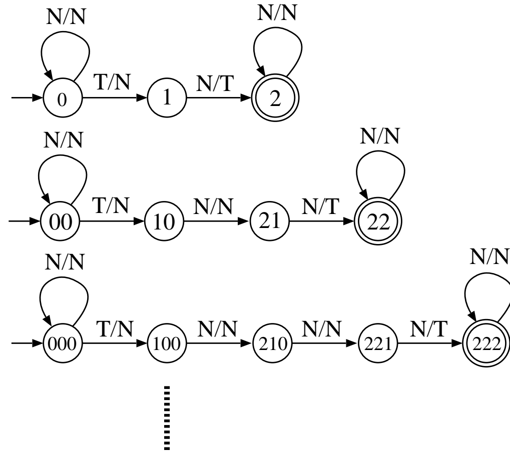


Fig. 3.4. The history transducer for the token passing protocol from Fig. 3.1

simulations on states.[5] The basic idea here is that we can consider two states q_1 and q_2 of a transducer (or automaton) A equal if there are states q, q' of A such that q backward simulates q_1 and forward simulates q_2 and q' backward simulates q_2 and forward simulates q_1. Then, by collapsing q_1 and q_2, which joins the paths leading to q_1 and continuing from q_2 in A (and vice-versa), we do not change the language of the automaton (transducer).

In [AdJN03, Nil05], one particular column equivalence is introduced. It is based on the notion of the so-called *left-copying* and *right-copying* transducer states. A state q_L of a transducer τ is called left-copying if all paths leading from the initial state of τ to q_L contain "copying" transitions of the form a/a only. Similarly, a state q_R of a transducer τ is called right-copying if all paths leading from q_R to some accepting state of τ are of the a/a form only. In our token passing example, the state 0 is left-copying, and 2 is right-copying.

It can be shown [AdJN03, Nil05] that for an arbitrary left-copying state q_L and a right-copying state q_R, the set $\{q_L, q_R\}^*$ can be partitioned into at most 7 equivalence classes. Consequently, in the columns, one can abstract the precise number of occurrences of left/right-copying states and also limit the number of their alternations to at most 3. Moreover, [AdJN03, Nil05] show that after a suitable pre-processing of a transducer using the so-called *bi-determinisation* (producing a transducer whose sub-automaton consisting of left-copying states is deterministic and whose sub-automaton consisting of

[5] In a length-preserving transducer $\tau = (Q, \Sigma, \delta, q_0, F)$, a state $q_2 \in Q$ forward simulates a state $q_1 \in Q$ iff for every $a, b \in \Sigma$ and $q_1' \in Q$ such that $q_1 \xrightarrow[\tau]{a/b} q_1'$ there is a state $q_2' \in Q$ such that $q_2 \xrightarrow[\tau]{a/b} q_2'$, q_2' forward simulates q_1', and if $q_1' \in F$, then also $q_2' \in F$. The notion of the backward simulation is analogous.

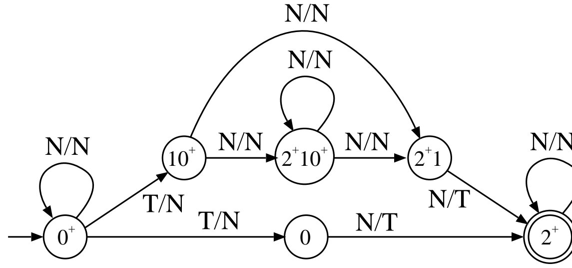


Fig. 3.5. A column-based transducer encoding the reachability relation ϱ^+ of the token passing protocol from Fig. 3.1

right-copying states is reverse-deterministic), it is not necessary to consider columns with successive appearances of distinct left/right-copying states—such states become either not reachable or not productive.

Having a suitable column equivalence, one could transform τ_{hist} into a hopefully finite quotient transducer $\tau_{hist/\simeq}$. However, τ_{hist} can never be obtained. The first solution to this problem was proposed in the older work [BJNT00]. It is based on a modification of the *subset construction* that is normally used for determinisation. One starts with the state q_0^+, computes all its successors under all pairs a/a' (which can be done by transducing the columns by special transducers built for every a/a' from the original transducer τ), *saturates* the obtained states by ignoring the number of left-copying states, and repeats the process from the newly generated states till a fix-point is reached.

A newer approach first proposed in [AdJN02] is to replace the history transducer by instead computing its underapproximations $\tau_{\leq k}$ obtained by a finite number k of compositions of τ. These approximations can be obtained *incrementally* by looking for transitions $x \xrightarrow[\tau_{\leq k}]{a/a'} x'$ and $q \xrightarrow[\tau]{a'/a''} q'$ and adding a new transition $xq \xrightarrow[\tau_{\leq k+1}]{a/a''} x'q'$. Adding of new transitions can then be combined with *quotienting* of the set of states (i.e., with merging of equivalent states) with respect to the proposed column equivalence. This process goes on till a fixpoint is reached. This way, in our token passing example, we obtain the reachability relation ϱ^+ encoded by the transducer shown in Fig. 3.5.

The presented techniques can also be adapted for computing directly the reachability set. Moreover, a sufficient condition under which the construction is guaranteed to terminate is known—it is the so-called *bounded local depth* condition [JN00, BJNT00, Nil05]. Intuitively, this condition requires that the single-step transition relation allows each position in a word to be changed a bounded number of times only (in the token passing example, the bound is 2, and so the construction is guaranteed to terminate).

3.4.3 Extrapolation

The *extrapolation* (or *widening*) approach to regular model checking was first proposed in [BJNT00]. It is based on comparing successive elements of the sequence I, $\varrho(I)$, $\varrho(\varrho(I))$, ..., trying to find some repeated growth pattern in this sequence, and adding an arbitrary number of occurrences of such a pattern into the reachability set.

In [BJNT00], the following technique was in particular proposed. Let $L \subseteq \Sigma^*$ be a so-far computed reachability set and $\varrho \subseteq \Sigma^* \times \Sigma^*$ be a regular one-step transition relation. One checks—e.g., by examining the structure of the finite-state automata encoding the involved regular sets—whether there are regular sets L_1, L_2, and Δ such that the following two conditions hold:

C1. $L = L_1.L_2$ and $\varrho(L) = L_1.\Delta.L_2$ and
C2. $L_1.\Delta^*.L_2 = \varrho(L_1.\Delta^*.L_2) \cup L$.

If the conditions hold, $L_1.\Delta^*.L_2$ is added to the so-far computer reachability set.

Intuitively, Condition C1 means that the effect of applying ϱ is to add Δ between L_1 and L_2. On the other hand, Condition C2 ensures that $\varrho^*(L) \subseteq L_1.\Delta^*.L_2$, and so we add at least all the configurations reachable from L by iterating ϱ. Note that the exactness of the acceleration—i.e., whether $L_1.\Delta^*.L_2 \subseteq \varrho^*(L)$ holds too—is not guaranteed in general. However, [BJNT00] gives a sufficient condition on ϱ under which Conditions C1 and C2 lead to an exact acceleration. This condition in particular requires ϱ to be a *well-founded* relation not allowing any word to have an infinite number of predecessors wrt. ϱ. In [BJNT00], the authors also give a syntactic criterion for the so-called *simple rewriting relations* that are guaranteed to satisfy this condition and that seem to appear quite often in practice—we refer an interested reader to [BJNT00] for more details.

In our token passing example protocol, we get $L = I = T.N^*$ and $\varrho(L) = N.T.N^*$ (cf. Fig. 3.1(a) and 3.2). We can choose $L_1 = \varepsilon$, $\Delta = N$, and $L_2 = T.N^*$, and we immediately get the reachability set $N^*.T.N^*$ from Fig. 3.3(a).

In [Tou01], the described extrapolation (widening) principle is extended to allow for several growths in a word and for different growths in different contexts. The technique can also be applied in a similar way to compute the reachability relation, i.e., to accelerate the sequence of compositions of ϱ. Moreover, [Tou01] shows how to nest the widening principle and compute the effect of iterating a sequence of relations.

The results of [BJNT00, Tou01] on extrapolation-based acceleration have been further improved in [BLW03]. First of all, [BLW03] proposed an improved technique for efficiently detecting a growth pattern in reachability sets encoded by finite-state automata. The detection is based on comparing two minimal deterministic automata A_1 and A_2 and splitting A_1 to a head part and a tail part and A_2 to a head part isomorphic to the head part of A_1, a growth, and a tail part isomorphic to the tail part of A_1.

For separating the head, growth, and tail parts, a comparison of states of A_1 and A_2 wrt. the so-called *forward* and *backward language equivalence* is used. Two states—one from A_1 and another from A_2—are forward language equivalent if they accept the same languages. Two such states are backward language equivalent if their backward languages are equal. The head parts of the automata then consist of backward language equivalent states, and the tail parts of forward language equivalent states. The forward language equivalence can be efficiently checked using the standard Hopcroft's minimisation procedure, and the backward language equivalence—taking into account the automata are deterministic minimal—by a simultaneous search through the initial parts of the automata identifying their isomorphic initial sub-automata.

The authors of [BLW03] then also propose a way of how to extrapolate using the detected growth (i.e., how to add some "looping" arcs on the growth into the automata), how to check that such an extrapolation is safe, and further a decidable (though relatively expensive) sufficient preciseness criterion. Moreover, the authors observe that it is sometimes useful to compare not the immediate successors in the sequence I, $\varrho(I)$, $\varrho(\varrho(I))$, ..., but the elements that appear at certain sample distances. For instance, when analysing systems with counters encoded as NDDs, it is useful to sample at distances that are a power of 2, which is related to the binary encoding of the integer vectors in NDDs. The technique has been implemented on top of the LASH automata libraries [LAS].

3.5 Abstract Regular Model Checking

A crucial problem to be faced in regular model checking is the *state space explosion in automata (transducer) representations* of the sets of configurations (or reachability relations) being examined. One of the sources of this problem is related to the nature of the previously mentioned regular model checking techniques. Typically, these techniques try to calculate the *exact* reachability sets (or relations) independently of the property being verified. However, it would often be enough to only compute an overapproximation of the reachability set (or relation) precise enough just to verify the given property of interest. Indeed, as we have already said in the introduction of the thesis, this is the way large (or infinite) state spaces are often being successfully handled outside the domain of regular model checking using the so-called *abstract-check-refine* paradigm often implemented in the form of the *counterexample guided abstraction refinement* (CEGAR) loop [GS97, BLO98, Sai00, CGJ$^+$00a, HJMS02, DD02].

CEGAR is, e.g., embedded in tools for software model checking like Slam [BR01], Magic [CCG$^+$04], or Blast [HJMS03]. All these tools use the method of *predicate abstraction* [GS97] where a finite set of boolean predicates is used to abstract a concrete system C into an abstract one A by considering equivalent the configurations of C that satisfy the same predicates. If a property is

verified in A, it is guaranteed to hold in C too. If a counterexample is found in A, one can check if it is also a counterexample for C. If not, this *spurious* counterexample can be used to *refine* the abstraction such that the new abstract system A' no longer admits the spurious counterexample. In this way, one can construct finer and finer abstractions until a sufficient precision is achieved and the property is verified, or a real counterexample is found.

Inspired by the above considerations, we proposed in [BHV04] a new approach to regular model checking based on the abstract-check-refine paradigm. Instead of precise acceleration techniques, we use abstract fixpoint computations in some *finite* domain of automata. The abstract fixpoint computations always terminate and provide overapproximations of the reachability sets (relations). To achieve this, we define techniques that systematically map any automaton M to an automaton M' from some finite domain such that M' recognises a superset of the language of M. For the case that the computed overapproximation is too coarse and a spurious counterexample is detected, we provide effective principles allowing the abstraction to be refined such that the new abstract computation does not encounter the same counterexample.

We propose here two techniques for abstracting automata, and two further abstractions are then described in Chapter 4 where they are inspired by the special needs of using abstract regular model checking in verification of programs with dynamic data structures. The abstractions we discuss in this section take into account the structure of automata and are based on collapsing their states according to some equivalence relation. The first one is inspired by predicate abstraction. However, notice that contrary to the classical predicate abstraction, we associate predicates with states of automata representing sets of configurations rather than with the configurations themselves. An abstraction is defined by a set of regular *predicate languages* L_P. We consider a state q of an automaton M to "satisfy" a predicate language L_P if the intersection of L_P with the language $L(M, q)$ accepted from the state q is not empty. Subsequently, two states are equivalent if they satisfy the same predicates. The second abstraction technique we propose is then based on considering two automata states equivalent if their *languages of words up to a certain fixed length* are equal. For both of these two abstraction methods, we provide effective refinement techniques allowing us to discard spurious counterexamples.

We also introduce several natural alternatives to the basic approaches based on backward and/or trace languages of states of automata. For them, it is not always possible to guarantee the exclusion of a spurious counterexample, but according to our experience, they still provide good practical results.

All of our techniques can be applied to dealing with reachability sets (obtained by iterating length-preserving or even general transducers) as well as length-preserving reachability relations.

We have implemented the different abstraction and refinement schemas in a prototype tool and tested them on a number of examples of various types of systems including parametric networks of processes, pushdown systems,

counter automata, systems with queues, and—for the first time in the context of regular model checking—a program manipulating dynamic linked data structures (in particular, we considered the common list reversion procedure). The experiments show that our techniques are quite powerful in all the considered cases and that they are complementary—different techniques turn out to be the most successful in different scenarios. The results are very promising and compare very favourably with other existing tools. The success in verification of the list reversion procedure then made us develop this field much further (including a more systematic encoding of programs manipulating dynamic linked data structures and more efficient specialised abstractions) as discussed in Chapter 4.

Before proceeding to the details of our technique, let us recall that for verification of parameterised networks of processes, several other methods using abstractions have been proposed [BBLS00, PXZ02]. Contrary to our approach, these methods do not provide the possibility of refinement of the abstraction. Moreover, they are specialised for parameterised networks whereas our technique is generic.

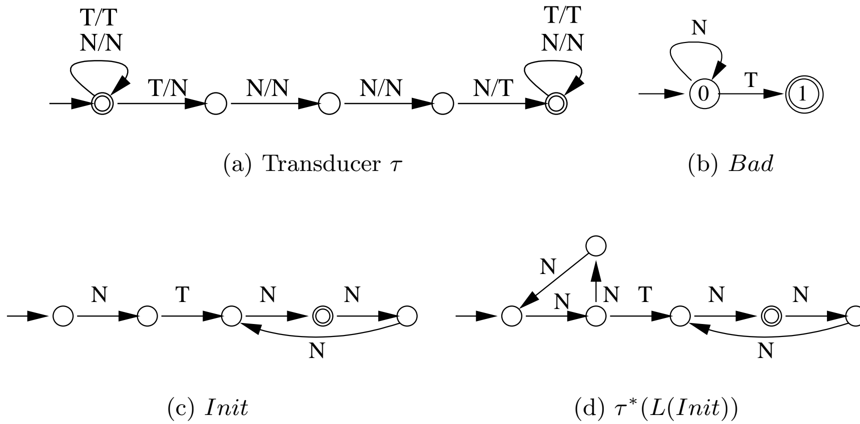
In the rest of the section, we first introduce a simple example that we use to demonstrate the techniques, and we also add some basic assumptions we make about the automata notions we use. Then, we introduce the general framework of abstract regular model checking and instantiate it with two concrete abstraction principles. Next, we discuss experiments we performed to illustrate capabilities of our method.

3.5.1 A Running Example and Some Basic Assumptions

As a simple running example capable of illustrating the different techniques we propose for abstract regular model checking, we consider a slight modification of the token passing protocol from Fig. 3.1. The modification consists in that each process can pass the token to its *third* right neighbour (instead of its direct right neighbour). The one-step transition relation of the system is encoded by the transducer τ in Fig. 3.6 (a). The transducer includes the identity relation too. In the initial configurations described by the automaton $Init$ from Fig. 3.6 (c), the second process has the token, and the number of processes is divisible by three. We want to show that it is not possible to reach any configuration where the last process has the token. This set is described by the automaton Bad from Fig. 3.6 (b).

Note that in the following, in order to shorten the descriptions, we *identify a transducer and the relation it represents* and write $\tau(L)$ instead of $\varrho(\tau)(L)$. Let $\iota \subseteq \Sigma^* \times \Sigma^*$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations (transducers) $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$, and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. As in our running example, we suppose $\iota \subseteq \tau$ for the rest of the section meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

The properties we want to check in abstract regular model checking are primarily reachability properties. Given a system with a transition relation



(a) Transducer τ (b) *Bad*

(c) *Init* (d) $\tau^*(L(Init))$

Fig. 3.6. A transducer τ modelling a modified token passing protocol and automata describing the initial, bad, and reachable configurations of the system

modelled as a transducer τ, a regular set of initial configurations given by an automaton *Init*, and a set of "bad" configurations given by an automaton *Bad*, we want to check $\tau^*(L(Init)) \cap L(Bad) = \emptyset$. We transform more complicated properties into reachability by composing the appropriate property automaton with the system being checked. In this way, even liveness properties may be handled if the transition relation is instrumented to allow for loop detection as we have already mentioned in Section 3.3 (we will return to this in Section 3.5.5). For our running example, $\tau^*(L(Init))$ is shown in Fig. 3.6(d), and the property of interest clearly holds. However, as we have also already said, in general, $\tau^*(L(Init))$ is neither guaranteed to be regular nor computable. In the following, the verification task is thus to find a regular overapproximation $L \supseteq \tau^*(L(Init))$ such that $L \cap L(Bad) = \emptyset$.

3.5.2 The Method of Abstract Regular Model Checking

We now describe the general approach of abstract regular model checking and propose a common framework for automata abstraction based on collapsing states of the automata. This framework is then instantiated in several concrete ways in the following two sections (and two further, more specialised abstractions are described also in Chapter 4). We concentrate on the use of abstract regular model checking for dealing with reachability sets. However, the techniques we propose may be applied to dealing with reachability relations too—though in the context of length-preserving transducers only. (Indeed, length-preserving transducers over an alphabet Σ can be seen as finite-state automata over $\Sigma \times \Sigma$.) We illustrate the applicability of the method to deal-

ing with reachability relations by one of the experiments presented in Section 3.5.5.

The Basic Framework of Automata Abstraction

Let Σ be a finite alphabet and $\mathbb{M}_\Sigma$ the set of all finite automata over Σ. By an *automata abstraction function* α, we understand a function that maps every automaton M over Σ to an automaton $\alpha(M)$ whose language is an overapproximation of the one of M. To be more precise, for some abstract domain of automata $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$, α is a mapping $\mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ such that $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. We call α *finitary* iff its range $\mathbb{A}_\Sigma$ is finite.

Working conveniently on the level of automata, given a transition relation expressed as a transducer τ over Σ and an automata abstraction function α, we introduce the *abstract transition function* τ_α as follows: For each automaton $M \in \mathbb{M}_\Sigma$, $\tau_\alpha(M) = \alpha(\hat{\tau}(M))$ where $\hat{\tau}(M)$ is the minimal deterministic automaton of $\tau(L(M))$. Now, we may iteratively compute the sequence $(\tau_\alpha^i(M))_{i\geq 0}$. Since we suppose $\iota \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(M) = \tau_\alpha^k(M)$. The definition of α implies $L(\tau_\alpha^k(M)) \supseteq \tau^*(L(M))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(M))$.

Refining Automata Abstractions

We call an automata abstraction function α' a *refinement* of α iff $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Moreover, we call α' a *true refinement* iff it yields a smaller overapproximation in at least one case—formally, iff $\exists M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subset L(\alpha(M))$.

A need to refine α arises when a situation depicted in Fig. 3.7 happens. Suppose we are checking whether no configuration from the set described by some automaton *Bad* is reachable from some given set of initial configurations described by an automaton M_0. We suppose $L(M_0) \cap L(Bad) = \emptyset$—otherwise the property being checked is broken already by the initial configurations. Let $M_0^\alpha = \alpha(M_0)$ and for each $i > 0$, $M_i = \hat{\tau}(M_{i-1}^\alpha)$ and $M_i^\alpha = \alpha(M_i) = \tau_\alpha(M_{i-1}^\alpha)$. There exist k and l $(0 \leq k < l)$ such that: (1) $\forall i : 0 \leq i < l : L(M_i) \cap L(Bad) = \emptyset$. (2) $L(M_l) \cap L(Bad) = L(X_l) \neq \emptyset$. (3) If we define X_i as the minimal deterministic automaton accepting $\tau^{-1}(L(X_{i+1})) \cap L(M_i^\alpha)$ for all i such that $0 \leq i < l$, then $\forall i : k < i < l : L(X_i) \cap L(M_i) \neq \emptyset$ and $L(X_k) \cap L(M_k) = \emptyset$ despite $L(X_k) \neq \emptyset$. Next, we see that either $k = 0$ or $L(X_{k-1}) = \emptyset$, and it is clear that we have encountered a *spurious counterexample*.

Note that when no l can be found such that $L(M_l) \cap L(Bad) \neq \emptyset$, the computation eventually reaches a fixpoint, and the property is proved to hold. On the other hand, if $L(X_0) \cap L(M_0) \neq \emptyset$, we have proved that the property is broken.

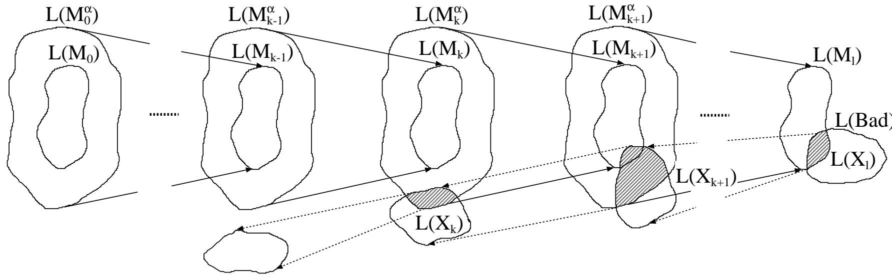


Fig. 3.7. A spurious counterexample in an abstract regular fixpoint computation

The *spurious counterexample may be eliminated* by refining α to α' such that for any automaton M whose language is disjoint with $L(X_k)$, the language of its α'-abstraction will not intersect $L(X_k)$ either. Then, the same faulty reachability computation (i.e., the same sequence of M_i and M_i^α) may not be repeated because we exclude the abstraction of M_k to M_k^α. Moreover, the reachability of the bad configurations is in general excluded unless there is another reason for it than overapproximating by subsets of $L(X_k)$.

A slightly *weaker way of eliminating the spurious counterexample* consists in refining α to α' such that at least the language of the abstraction of M_k does not intersect with $L(X_k)$. In such a case, it is not excluded that some subset of $L(X_k)$ will again be used for an overapproximation somewhere, but we still exclude a repetition of exactly the same faulty computation. The obtained refinement can be coarser, which may lead to more refinements and a slower computation. On the other hand, the computation may terminate sooner due to quickly jumping to the fixpoint and use less memory due to working with less structured sets of configurations of the systems being verified—the abstraction is prevented from becoming unnecessarily precise in this case. For the latter reason, as illustrated later, one may sometimes successfully use even some more heuristic approaches that guarantee that the spurious counterexample will only eventually be excluded (i.e., after a certain number of refinements) or that do not guarantee the exclusion at all.

An obvious danger of using a heuristic approach that does not guarantee an exclusion of spurious counterexamples is that the computation may easily start looping. Notice, however, that even when we refine automata abstractions such that spurious counterexamples are always excluded, and the computation does not loop, we do not guarantee that it will eventually stop—we may keep refining forever. Indeed, the verification problem we are solving is undecidable in general.

Abstracting Automata by Collapsing Their States

In the following two sections, we discuss several concrete automata abstraction functions. They are based on automata state equivalence schemas that

define for each automaton from $\mathbb{M}_\Sigma$ an equivalence relation on its states. An automaton is then abstracted by collapsing all its states related by this equivalence. We suppose such an equivalence to reflect the fact that the future and/or history of the states to be collapsed is close enough, and the difference may be abstracted away.

Formally, an *automata state equivalence schema* $\mathbb{E}$ assigns an automata state equivalence $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ to each finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ over Σ. We define the *automata abstraction function* $\alpha_{\mathbb{E}}$ based on $\mathbb{E}$ such that $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ *finitary* iff $\alpha_{\mathbb{E}}$ is finitary. We *refine* $\alpha_{\mathbb{E}}$ by refining $\mathbb{E}$ such that more states are distinguished in at least some automata.

The automata state equivalence schemas presented below are then all based on one of the following two basic principles: (1) comparing states wrt. the intersections of their forward/backward languages with some *predicate languages* (represented by the appropriate *predicate automata*) and (2) comparing states wrt. their forward/backward behaviours up to a certain *bounded length*.

3.5.3 Automata State Equivalences Based on Predicate Languages

The two automata state equivalence schemas we introduce in this section—$\mathbb{F}_\mathcal{P}$ based on forward languages of states and $\mathbb{B}_\mathcal{P}$ based on backward languages—are both defined wrt. a finite set of *predicate automata* $\mathcal{P}$. They compare two states of a given automaton according to the intersections of their forward/backward languages with the languages of the predicates. Below, we first introduce the basic principles of the schemas and then add some implementation and optimisation notes.

The $\mathbb{F}_\mathcal{P}$ Automata State Equivalence Schema

The automata state equivalence schema $\mathbb{F}_\mathcal{P}$ defines two states of a given automaton to be equivalent when their languages have a *nonempty intersection with the same predicates* of $\mathcal{P}$. Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_\mathcal{P}$ defines the state equivalence as the equivalence $\sim_M^\mathcal{P}$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^\mathcal{P} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Clearly, as $\mathcal{P}$ is finite and there is only a finite number of subsets of $\mathcal{P}$ representing the predicates with which a given state has a nonempty intersection, $\mathbb{F}_\mathcal{P}$ is *finitary*.

For our example from Fig. 3.6, if we take as $\mathcal{P}$ the automata of the languages of the states of *Bad*, we obtain the automaton in Fig. 3.8(a) as the abstraction of *Init* from Fig. 3.6(c). This is because all states of *Init* except the final one become equivalent. Then, the intersection of $\hat{\tau}(\alpha(Init))$ with the bad configurations—shown in Fig. 3.8(c)—is not empty, and we have to refine the abstraction.

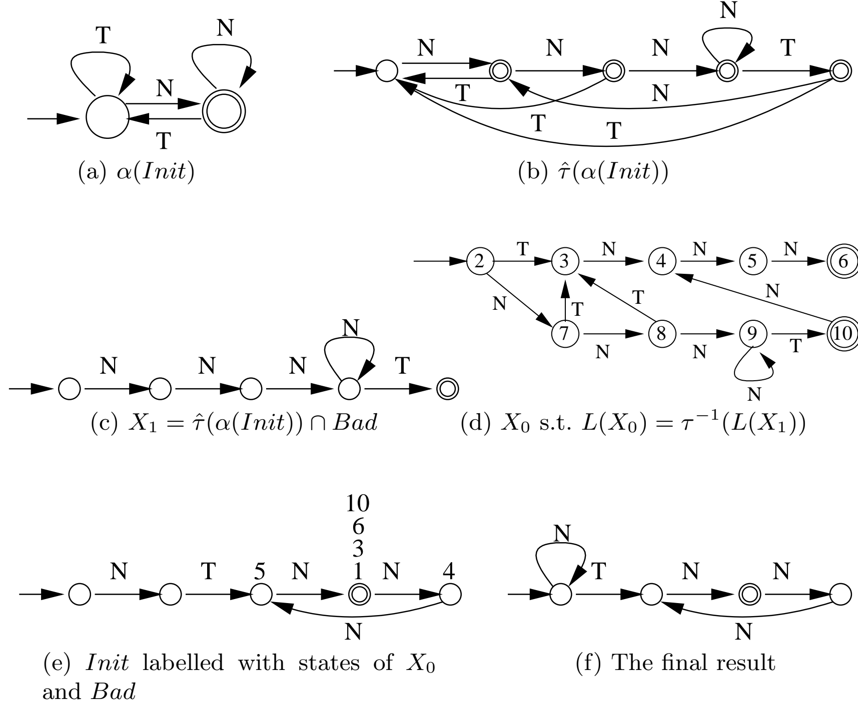


Fig. 3.8. An example using abstraction based on predicate languages

The $\mathbb{F}_{\mathcal{P}}$ schema may be *refined by adding new predicates* into the current set of predicates $\mathcal{P}$. In particular, we can extend $\mathcal{P}$ by automata corresponding to the languages of all the states in X_k from Fig. 3.7. Theorem 3.5.1 shows that this prevents abstractions of languages disjoint with $L(X_k)$, such as—but not only—$L(M_k)$, from intersecting with $L(X_k)$. Consequently, as we have already explained, a repetition of the same faulty computation is excluded, and the set of bad configurations will not be reached unless there is another reason for this than overapproximating by subsets of $L(X_k)$.

Theorem 3.5.1 *Let $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ be any two finite automata and let $\mathcal{P}$ be a finite set of predicate automata such that $\forall q_X \in Q_X : \exists P \in \mathcal{P} : L(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{F}_{\mathcal{P}}}(M)) \cap L(X) = \emptyset$ too.*

Proof. We prove the theorem by contradiction. Suppose $L(\alpha_{\mathbb{F}_{\mathcal{P}}}(M)) \cap L(X) \neq \emptyset$. Let $w \in L(\alpha_{\mathbb{F}_{\mathcal{P}}}(M)) \cap L(X)$. As w is accepted by $\alpha_{\mathbb{F}_{\mathcal{P}}}(M)$, M must accept it when we allow it to perform a certain number of "jumps" between states equal wrt. $\sim_M^{\mathcal{P}}$—after accepting a prefix of w and getting to some $q \in Q_M$,

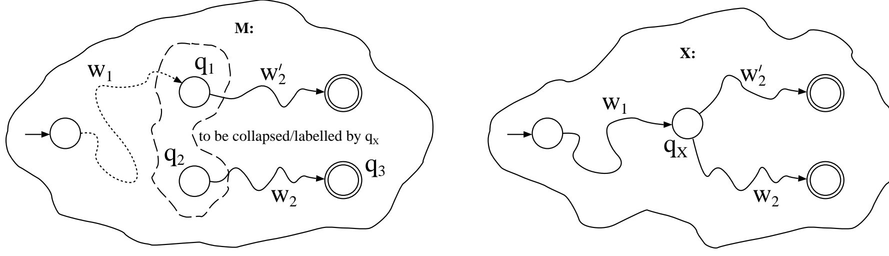


Fig. 3.9. An illustration of the proof of Theorem 3.5.1

M is allowed to jump to any $q' \in Q_M$ such that $q \sim_M^{\mathcal{P}} q'$ and go on accepting from there (with or without further jumps).

Suppose that the minimum number of jumps needed to accept a word from $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$ in M is i, $i > 0$, and let w' be such a word. Let the last jump within accepting w' in M be from some state $q_1 \in Q_M$ to some $q_2 \in Q_M$ such that $q_1 \sim_M^{\mathcal{P}} q_2$. Let $w' = w_1 w_2$ such that w_1 is accepted (possibly with jumps) just before the jump from q_1 to q_2 (cf. Fig. 3.9). Clearly, $q_2 \xrightarrow[M]{w_2} q_3$ for some $q_3 \in F_M$. We know that X accepts w'. Suppose that after accepting w_1, it is in some $q_X \in Q_X$. As $w_2 \in L(X, q_X)$ and $w_2 \in L(M, q_2)$, $L(M, q_2) \cap L(P) \neq \emptyset$ for the predicate(s) $P \in \mathcal{P}$ for which $L(P) = L(X, q_X)$. Moreover, as $q_1 \sim_M^{\mathcal{P}} q_2$, $L(M, q_1) \cap L(P) \neq \emptyset$ too. This implies there exists $w_2' \in L(P)$ such that $w_2' \in L(M, q_1)$ and $w_2' \in L(X, q_X)$. However, this means that $w_1 w_2' \in L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$ can be accepted in M with $i - 1$ jumps, which is a contradiction to the assumption of i being the minimum number of jumps needed. $\square$

In our example, we refine the abstraction by extending $\mathcal{P}$ with the automata representing the languages of the states of X_0 from Fig. 3.8(d). Fig. 3.8(e) then indicates for each state q of $Init$, the predicates corresponding to the states of Bad and X_0 whose languages have a non-empty intersection with the language of q. The first two states of $Init$ are equivalent and are collapsed to obtain the automaton from Fig. 3.8(f), which is a fixpoint showing that the property is verified. Notice that it is an overapproximation of the set of reachable configurations from Fig. 3.6(d).

The price of refining $\mathbb{F}_\mathcal{P}$ by adding predicates for all the states in X_k may seem prohibitive, but fortunately this is not the case in practice. As described later on in this section, we do not have to treat all the new predicates separately. We exploit the fact that they come from one original automaton and share large parts of their structure. In fact, we can work just with the original automaton and each of its states may be considered an initial state of some predicate. This way, adding the original automaton as the only predicate and adding predicates for all of its states becomes roughly equal. Moreover, the refinement may be weakened by taking into account just some states of X_k as discussed later on.

The $\mathbb{B}_{\mathcal{P}}$ Automata State Equivalence Schema

The $\mathbb{B}_{\mathcal{P}}$ automata state equivalence schema is an alternative of $\mathbb{F}_{\mathcal{P}}$ based on *backward languages of states* rather than the forward ones. For an automaton $M = (Q, \Sigma, \delta, q_0, F)$, it defines the state equivalence as the equivalence $\overleftarrow{\sim}_M^{\mathcal{P}}$ such that $\forall q_1, q_2 \in Q : q_1 \overleftarrow{\sim}_M^{\mathcal{P}} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap \overleftarrow{L}(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap \overleftarrow{L}(M, q_2) \neq \emptyset)$.

Clearly, $\mathbb{B}_{\mathcal{P}}$ is *finitary* for the same reason as $\mathbb{F}_{\mathcal{P}}$. It may also be *refined* by extending $\mathcal{P}$ by automata corresponding to the languages of all the states in X_k from Fig. 3.7. As stated in Theorem 3.5.2, the effect is the same as for $\mathbb{F}_{\mathcal{P}}$.

Theorem 3.5.2 *Let $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ be any two finite automata and let $\mathcal{P}$ be a finite set of predicate automata such that $\forall q_X \in Q_X : \exists P \in \mathcal{P} : \overleftarrow{L}(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{B}_{\mathcal{P}}}(M)) \cap L(X) = \emptyset$ too.*

Proof. The theorem can be proved by contradiction in a similar way as Theorem 3.5.1. This time, as a consequence of working with backward languages of states, we do not deal with the last jump, but the first jump in accepting some $w' \in L(\alpha_{\mathbb{B}_{\mathcal{P}}}(M)) \cap L(X)$ in M. We do not look for a replacement w_2' of w_2 to be accepted from q_1 instead of q_2, but for a replacement w_1' of w_1 to be accepted before q_2 rather than before q_1. □

Implementing and Optimising Collapsing Based on $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$

The abstraction of an automaton M wrt. the automata state equivalence schema $\mathbb{F}_{\mathcal{P}}$ may be implemented by first labelling states of M by the states of predicate automata in $\mathcal{P}$ with whose languages they have a non-empty intersection and then collapsing the states of M that are labelled by the initial states of the same predicates. (Provided the sets of states of the predicate automata are disjoint.) The labelling can be done in a way similar to constructing a backward synchronous product of M with the particular predicate automata: (1) $\forall P \in \mathcal{P}\ \forall q_F^P \in F_P\ \forall q_F^M \in F_M$: q_F^M is labelled by q_F^P, and (2) $\forall P \in \mathcal{P}\ \forall q_1^P, q_2^P \in Q_P\ \forall q_1^M, q_2^M \in Q_M$: if q_2^M is labelled by q_2^P, and there exists $a \in \Sigma$ such that $q_1^M \xrightarrow[\delta_M]{a} q_2^M$ and $q_1^P \xrightarrow[\delta_P]{a} q_2^P$, then q_1^M is labelled with q_1^P. The abstraction of an automaton M wrt. the $\mathbb{B}_{\mathcal{P}}$ schema may be implemented analogously.

If the above construction is used, it is then clear that when *refining* $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$, we can just add X_k into $\mathcal{P}$ and modify the construction such that in the collapsing phase, we simply take into account all the labels by states of X_k and do not ignore the (anyway constructed) labels other than $q_0^{X_k}$.

Moreover, we can try to optimise the refinement of $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ by replacing X_k in $\mathcal{P}$ by its *important tail/head part* defined wrt. M_k as the subautomaton of X_k based on the states of X_k that appear in at least one of the labels of M_k wrt. $\mathbb{F}_{\mathcal{P}\cup\{X_k\}}/\mathbb{B}_{\mathcal{P}\cup\{X_k\}}$, respectively. As stated in Theorem 3.5.3, the effect of such a refinement corresponds to the weaker way of refining automata abstraction functions described in Section 3.5.2. This is due to the strong link of the important tail/head part of X_k to M_k wrt. which it is computed. A repetition of the same faulty computation is then excluded, but the obtained abstraction is coarser, which may sometimes speed up the computation as we have already discussed.

Theorem 3.5.3 *Let M and X be any finite automata over Σ and $Y = (Q_Y, \Sigma, \delta_Y, q_0^Y, F_Y)$ the important tail/head part of X wrt. $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ and M. If $\mathcal{P}'$ is such that $\forall q_Y \in Q_Y \; \exists P \in \mathcal{P}' : L(Y, q_Y)/\overleftarrow{L}(Y, q_Y) = L(P)$ and $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{F}_{\mathcal{P}'}/\mathbb{B}_{\mathcal{P}'}}(M)) \cap L(X) = \emptyset$.*

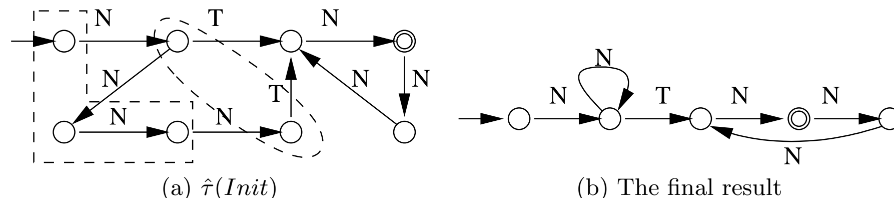
Proof. The key last/first jump in an accepting run of M mentioned in the proofs of Theorems 3.5.1, 3.5.2 is between states that can be labelled by some states of X. The concerned states of X are thus in the important tail/head part of X, and the proof construction of Theorems 3.5.1, 3.5.2 can still be applied. □

A further possible heuristic to optimise the refinement of $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ is trying to find just one or two *key states* of the important tail/head part of X_k such that if their languages are considered in addition to $\mathcal{P}$, $L(M_k^\alpha)$ will not intersect $L(X_k)$.

We close the section by noting that in the *initial set of predicates* $\mathcal{P}$ of $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$, we may use, e.g., the automata describing the set of bad configurations and/or the set of initial configurations. Further, we may also use the domains or ranges of the transducers encoding the particular transitions in the systems being examined (whose union forms the one-step transition relation τ which we iterate). The meaning of the latter predicates is similar to using guards or actions of transitions in predicate abstraction [BLO98].

3.5.4 Automata State Equivalences Based on Finite-Length Languages

We now present the possibility of defining automata state equivalence schemas based on comparing automata states wrt. a certain bounded part of their languages. It is a simple, yet (according to our practical experience) often quite efficient approach. As a basic representative of this kind of schemas, we first present the schema $\mathbb{F}_n^L$ based on forward languages of words of a limited length. Then, we discuss its possible alternatives.



(a) $\hat{\tau}(Init)$ (b) The final result

Fig. 3.10. An example using abstraction based on languages of words up to length n (for $n = 2$)

The $\mathbb{F}_n^L$ automata state equivalence schema defines two states of an automaton to be equal if their *languages of words of length up to a certain bound* n are identical. Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_n^L$ defines the state equivalence as the equivalence $\sim_M^n$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

$\mathbb{F}_n^L$ is clearly *finitary*. It may be *refined* by incrementally increasing the bound n on the length of the words considered. This way, as we work with minimal deterministic automata, we may achieve the weaker type of refinement described in Section 3.5.2. Such an effect is achieved when n is increased to be equal or bigger than the number of states in M_k from Fig. 3.7 minus one. In a minimal deterministic automaton, this guarantees that all states are distinguishable wrt. $\sim_M^n$, and M_k will not be collapsed at all.

In Fig. 3.10, we apply $\mathbb{F}_n^L$ to the example from Fig. 3.6. We choose $n = 2$. In this case, the abstraction of the $Init$ automaton is $Init$ itself. Fig. 3.10(a) indicates the states of $\hat{\tau}(Init)$ that have the same languages of words up to size 2 and are therefore equivalent. Collapsing them yields the automaton shown in Fig. 3.10(b) (after determinisation and minimisation), which is a fixpoint. Notice that it is a different overapproximation of the set of reachable configurations than the one obtained using $\mathbb{F}_{\mathcal{P}}$. If we choose $n = 1$, we obtain a similar result, but we need one refinement step of the above described kind.

Let us, however, note that according to our practical experience, the increment of n by $|Q_M| - 1$ may often be too big. Alternatively, one may use its fraction (e.g., one half), increase n by the number of states in X_k (or its fraction), or increase n just by one. In such cases, an immediate exclusion of the faulty run is not guaranteed, but clearly, such a computation will be *eventually excluded* because n will sooner or later reach the necessary value. The impact of working with abstractions refined in a coarser way is then like in the case of using $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$.

Regarding the *initial value of* n, one may use, e.g., the number of states in the automaton describing the set of initial configurations or the set of bad configurations, their fraction, or again just one.

As a natural alternative to dealing with forward languages of words of a limited length, we may also think of *backward languages of words* of a limited length and forward or backward *languages of traces* with a limited length. The

automata equivalence schemas $\mathbb{B}_n^L$, $\mathbb{F}_n^T$, and $\mathbb{B}_n^T$ based on them can be formally defined analogously to $\mathbb{F}_n^L$.

Clearly, all these schemas are *finitary*. Moreover, we can *refine* them in a similar way as $\mathbb{F}_n^L$. For $\mathbb{F}_n^T$ and $\mathbb{B}_n^T$, however, no guarantee of excluding a spurious counterexample may be provided. Using $\mathbb{F}_n^T$, e.g., we can never distinguish the last three states of the automaton in Fig. 3.10(b)—they all have the same trace languages. Thus, we cannot remember that the token cannot get to the last process. Nevertheless, despite this, our practical experience shows that the schemas based on traces are quite successful in practice.

3.5.5 Experiments with Abstract Regular Model Checking

We have implemented the ideas described above in a prototype tool written in YAP Prolog using the FSA library [vN04]. To demonstrate that abstract regular model checking is applicable to verification of a broad variety of systems, we tried to apply the tool to a number of different verification tasks. Currently, the techniques are being re-implemented using the LASH [LAS] and MONA [KM01] automata libraries.

The Types of Systems Verified

Parameterised Networks of Processes

We considered several slightly idealised *mutual exclusion algorithms* for an arbitrary number of processes (namely the Bakery, Burns, Dijkstra, and Szymanski algorithms in versions similar to [Nil00]). In most of these systems, the particular processes are finite-state. We encode global configurations of such systems by words whose length corresponds to the number of participating processes, and each letter represents the local state of some process. In the case of the Bakery algorithm where each process contains an unbounded ticket value, this value is not represented directly, but encoded in the ordering of the processes in the word.

We verified the mutual exclusion property of the algorithms, and for the Bakery algorithm, we verified that some process will always eventually get to the critical section (communal liveness) as well as that each individual process will always eventually get there (individual liveness) under suitable fairness assumptions. For checking liveness, we manually composed the appropriate Büchi automata with the system being verified. Loop detection was allowed by working with pairs of configurations consisting of a remembered potential beginning of a loop (fixed at a certain—randomly chosen—point of time) and the current configuration being further modified. Checking that a loop is closed then consisted in checking that a pair of the same configurations was reached. To encode the pairs of configurations using finite automata, we interleaved their corresponding letters.

Push-down Systems

We considered a simple system of *recursive procedures*—the plotter example from [EHRS00]. We verified a safety part of the original property of interest describing the correct order of plotter instructions to be issued. In this case, we use words to encode the contents of the stack.

Systems with Queues

We experimented with a model of the Alternating Bit Protocol (ABP) for which we checked correctness of the delivery order of the messages. A word encoding a configuration of the protocol contained two letters representing internal states of the communicating processes. Moreover, it contained the contents of the two *lossy communication channels* with a letter corresponding to each message. Let us note that in this case, as well as in the above and below cases, general (non-length-preserving) transducers were used to encode transitions of the systems.

Petri Nets, Systems with Counters

We examined a general *Petri net* with inhibitor arcs, which can be considered an example of a system with *unbounded counters* too. In particular, we modelled a Readers/Writers system extended with a possibility of dynamic creation/deletion of processes, for which we verified mutual exclusion between readers and writers and between multiple writers. We considered a correct version of the system as well as a faulty one, in which we omitted one of the Petri net arcs. Markings of places in the Petri net were encoded in unary, and the particular values were put in parallel. (Using this encoding, a marking of a net with places p and q, two tokens in p, and four in q would be encoded as $q|q|pq|pq$.) In some other examples of systems with counters (such as the Bakery algorithm for two processes with unbounded counters), we also successfully used a binary encoding of the counters like in NDDs [WB98].

Dynamic Linked Data Structures

We considered verification of a *procedure for reversing singly-linked lists* shown in Fig. 3.11. It was for the first time ever that regular model checking has been applied to such a task. The encoding used, which is described below, was a bit different from the more systematic one that we proposed in the follow-up work [BHMV05] discussed in Chapter 4. We present here the older encoding for a comparison how the subject has further been developed.

When abstracting the memory manipulated by the procedure, we focus on the cases where in the first n memory cells (we take the biggest n possible) there are at most two linked lists linking consecutive cells, the first list in a descending way and the second one in an ascending way. We represent configurations of the procedure as words over the following alphabet: list items

```
1: x = NULL;
2: while (list != NULL) {
3:     y = list → next;
4:     list → next = x;
5:     x = list;
6:     list = y;
7: }
8: list = x;
```

Fig. 3.11. Reversing a singly-linked list

are represented by symbols $\underline{i}$, left/right pointers by $<$/$>$, pointer variables are represented by their names (*list* is shortened to l), and $\underline{o}$ is used to represent the memory outside the list. Moreover, we use symbols $\underline{iv}$ (resp. $\underline{ov}$) to denote that v points to i (resp. outside the list). We use $|$ to separate the ascending and descending lists. Pointer variables pointing to null are not present in the configuration representations. A typical abstraction of the memory may then look like $\underline{i} < \underline{i} < \underline{i} \mid \underline{il} > \underline{i}\ \underline{ox}$ where the first list contains three items, the second one two, *list* points to the beginning of the second list, x points outside the two lists, and y points to null. Provided such an abstraction is used for the memory contents (prefixed with the current control line), it is not difficult to associate transducers to each statement of the procedure. For example, the transducer corresponding to the statement $list \rightarrow next := x$ at line 4 transforms a typical configuration $4\ \underline{i} < \underline{ix} \mid \underline{il} > \underline{iy} > \underline{i}\ \underline{o}$ to the configuration $5\ \underline{i} < \underline{ix} < \underline{il} \mid \underline{iy} > \underline{i}\ \underline{o}$ (the successor of the item pointed to by *list* is not anymore the one pointed to by y, but the one pointed to by x). Then, the transducer τ corresponding to the whole procedure is the union of the transducers of all the statements.

If the memory contents does not fit the above described form, we abstract it to a single word with the "don't know" meaning. However, when we start from a configuration like $1\ \underline{il} > \underline{i} > \underline{i}\ \underline{o}$ or $1\ \underline{i} < \underline{i} < \underline{il}\ \underline{o}$, the verification shows that such a situation does not happen. Via a symmetry argument exploiting the fact that the procedure never refers to concrete addresses, the results of the verification may then easily be generalised to lists with items stored at arbitrary memory locations.

By computing an abstraction of the reachability set $\tau^*(Init)$, we checked that the procedure outputs a list. Moreover, by computing an overapproximation of the reachability relation τ^* of the system, we checked that the output list is a reversion of the input one (modulo the fact that we consider a finite number of distinguishable list items). To speed up the computation, the reachability relation was restricted to the initial configurations, i.e., to $\iota_{Init} \circ \tau^*$ where ι_{Init} is the identity relation with the domain (and range) restricted to $L(Init)$.

Table 3.1. Some results of experimenting with abstract regular model checking while using the finite-length-languages-based abstractions

Experiment	$\mathbb{F}_n^L/\mathbb{F}_n^T/\mathbb{B}_n^L/\mathbb{B}_n^T$	$T_{best}^{\mathbb{F}_n^L/\mathbb{F}_n^T/\mathbb{B}_n^L/\mathbb{B}_n^T}$
Bakery	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert/2$	0.02
Bakery/comm. liv.	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert$	0.14
Bakery/ind. liv.	Fw, $\mathbb{F}_n^T$, 1	8.66
Bakery – counters	Bw, $\mathbb{B}_n^L$, $\lvert Q_{Bad}\rvert$	0.08
ABP	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.32
Burns	Fw, $\mathbb{B}_n^T$, 1	0.31
Dijkstra	Fw, $\mathbb{F}_n^T$, 1	1.75
PDS	Bw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.02
Petri net/Read. Wr.	Fw, $\mathbb{B}_n^T$, special n	21.07
Faulty PN/Rd. Wr.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert$	0.73
Szymanski	Fw, $\mathbb{B}_n^T$, 1	0.25
Rev. Lists	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2 + \lvert Q_{X_k}\rvert/2$	0.61
Rev. Lists/Transd.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2$	21.79

A Summary of the Results

The efficiency of using the $\mathbb{F}_n^L$, $\mathbb{F}_n^T$, $\mathbb{B}_n^L$, or $\mathbb{B}_n^T$ automata state equivalence schemas heavily depends on the *choice of the initial value of* n and the *strategy of increasing it.* In our experiments, we have tried $|Q_{Bad}|$, $|Q_{Bad}|/2$, $|Q_{Init}|$, $|Q_{Init}|/2$, and 1 as the initial value of n and $|Q_{M_k}|$, $|Q_{M_k}|/2$, $|Q_{X_k}|$, $|Q_{X_k}|/2$, and 1 as its increment. The results we obtained are summarised in Table 3.1. In the table, we always mention the scenario for which we obtained the shortest execution time.[6] We first say whether it was in a forward or backward computation (i.e., starting from the initial configurations or the "bad" configurations), then the automata state equivalence schema used, followed by the initial value of n, and if it was needed, the increment of n written behind a plus symbol. In the case of the Readers/Writers example, the time consumption was quite high, and we tried to iteratively find a value of n for which it was the best.

Similarly to the above, the efficiency of using the $\mathbb{F}_\mathcal{P}/\mathbb{B}_\mathcal{P}$ automata state equivalence schemas depends a lot on the choice of the *initial predicates.* As the basic initial predicates in our experiments, we considered using automata representing the set of bad or initial configurations. We used them alone or together with automata corresponding to the domains or ranges of the transducers encoding the particular transitions in the systems being examined. The scenarios that lead to the best results are listed in Table 3.2. The *heuristic*

[6] In some cases, a few scenarios gave a very similar result out of which just one is mentioned.

Table 3.2. Some results of experimenting with abstract regular model checking while using the predicate-based abstractions

Experiment	$\mathbb{F}_\mathcal{P}/\mathbb{B}_\mathcal{P}$	$T_{best}^{\mathbb{F}_\mathcal{P}/\mathbb{B}_\mathcal{P}}$
Bakery	Fw, $\mathbb{F}_\mathcal{P}$, $[Bad]$	0.02
Bakery/comm. liv.	Fw, $\mathbb{F}_\mathcal{P}$, $[Bad\|Grd]$	0.13
Bakery/ind. liv.	Fw, $\mathbb{F}_\mathcal{P}$, $[Bad]$, Key St.	19.41
Bakery – counters	Bw, $\mathbb{B}_\mathcal{P}$, $[Bad\|Grd]$	0.09
ABP	Fw, $\mathbb{B}_\mathcal{P}$, $[Init\|Grd]$	0.68
Burns	Fw, $\mathbb{B}_\mathcal{P}$, $[Bad]$	0.06
Dijkstra	Fw, $\mathbb{B}_\mathcal{P}$, $[Bad]$	0.73
PDS	Bw, $\mathbb{F}_\mathcal{P}$, $[Bad]$	0.02
Petri net/Read. Wr.	Fw, $\mathbb{B}_\mathcal{P}$, $[Bad\|Grd]$	5.86
Faulty PN/Rd. Wr.	Fw, $\mathbb{B}_\mathcal{P}$, $[Init\|Grd]$	0.81
Szymanski	Fw, $\mathbb{F}_\mathcal{P}$, $[Init\|Grd]$	0.55
Rev. Lists	Fw, $\mathbb{B}_\mathcal{P}$, $[Bad\|Grd\|Act]$	1.29
Rev. Lists/Transd.	Fw, $\mathbb{B}_\mathcal{P}$, $[Init\|Grd\|Act]$	42.60

optimisation of the refinements described in Section 3.5.3, had a very significant positive impact in the case of checking individual liveness in the Bakery example. In the other cases, the effect was neutral or negative.

The times presented in Tables 3.1 and 3.2 are in seconds and were obtained on a computer based on an Intel Pentium 4 processor at 1.7 GHz. They do not include the time needed for reading the input model. Taking into account that the tool used was an early prototype written in YAP Prolog using the FSA library [vN04][7], the results are very positive. For example, the Uppsala Regular Model Checker [AdJN03] took from about 8 to 11 seconds when applied to a comparable encoding of the Burns, Szymanski, and Dijkstra examples (and the situation does not seem to have changed too much as yet [Nil05]). Finally, Tables 3.1 and 3.2 also show that apart from cases where the approaches based on languages of words/traces up to a bounded length and the ones based on intersections with predicate languages are roughly equal, there are really cases where either the former or the latter approach is faster. This experimentally justifies our interest in both of the techniques.

As we have noted in the introduction to the chapter, for some of the classes of systems we considered, there exist various special purpose verification approaches, and the appropriate verification problems are sometimes even decidable (as, e.g., for push-down systems [BEM97, EHRS00, Sch02b] or lossy channel systems [BP96, ABJ98, AAB99]). However, we wanted to show that our approach is generic and can be uniformly applied to all these

[7] Prolog was chosen as a rapid, but still relatively efficient, prototyping environment.

systems. Moreover, in the future, with a new version of our tool, we would like to compare the performance of abstract regular model checking with the specialised approaches on large systems. We believe that while we can hardly outperform these algorithms in general, in some cases of systems with complex state spaces, our approach could turn out to be quite competitive due to not working with the exact representation of the state spaces, but their potentially much simpler approximations in which many details not important for the property being checked are ignored.

Among all the scenarios considered, we have noticed just a very few in which the computation was *diverging* (or seemed to diverge)—e.g., forward verification of ABP or the Szymanski algorithm using $\mathbb{B}_{\mathcal{P}}$ with the automaton representing the set of initial states as the only initial predicate. The fact that the computation is diverging may be clearly deduced, e.g., when new predicates representing languages like a, aa, aaa, ... are being added. In our case, the divergence disappeared when a different set of initial predicates was used. However, we have also successfully tried to remove it by *accelerating the counterexample analysis*, which is a step sometimes applied in predicate abstraction [BLO98] too. We tried to accelerate by a simple widening schema (detecting a constant growth in the number of states in several consecutive newly added predicates and adding a self-loop based on every transition of the last such predicate leaving its initial part common with the previous predicate). Sometimes, the divergence could also be resolved by combining the two automata state equivalence schemas presented here (namely by abstracting new predicates added wrt. $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ according to the equivalences based on languages of words/traces of a bounded length).

3.6 Inference of Regular Languages

In this section, we describe one further way of implementing regular model checking. The method we describe here has originally been described in [HV04, HV05]. It is based on using *inference of regular languages* extending the work of [FO97]. The approach is motivated by the observation that for infinite-state systems whose behaviour can be modelled using length-preserving transducers, there is a finite computation for obtaining all reachable configurations up to a certain length. These configurations may be considered as a sample of the reachable configurations of the given system. Then, methods that have been developed for inference of regular languages may be used to generalise the sample with the aim of obtaining the full reachability set or an overapproximation of it that is precise enough to prove the property of interest (if it holds). We in particular concentrate on using the Trakhtenbrot-Barzdin algorithm [TB73] as the inference algorithm, but we also briefly mention some other approaches we have tried.

As shown by our experiments, the method provides *good performance results*. At the same time, in contrast to many other existing regular model

checking methods, *termination is guaranteed* for *all* the systems whose set of reachable configurations is regular (including, e.g., lossy channel systems and push-down systems). Similar results can then be obtained for dealing with reachability relations instead of reachability sets too.

From the above, it is clear that we now primarily concentrate on systems that may be encoded using length-preserving transducers. Dealing with length-preserving transducers is, however, sufficient even for verification of safety properties of non-length preserving systems. In such a case, words encoding configurations may be in advance extended with special blank symbols to be consumed whenever new useful symbols are to be inserted. Moreover, as we show, our method can be extended to deal with non-length-preserving transducers too. Then, however, our termination guarantee does not hold (though in practice, the method still behaves well).

As we describe at the end of the section, we have implemented the method and tested it on a number of examples of different systems similar to the ones presented in Section 3.5.5 including parametric networks of processes, a pushdown system, systems with counters, a system with lossy queues, and a system with a linked list as a representative of systems with recursive data structures. The experiments show that the method is quite efficient with the results being often comparable with those of abstract regular model checking for which, however, termination guarantees are not very clear as yet.[8]

Before proceeding to the details of the method, let us add a few words on the related work. The idea of using inference of regular languages for regular model checking has already been used in [FO97], which, however, primarily targeted parameterised rings only, and the computation loop used there was different than ours and required a certain manual classification of the transitions of the systems used. Moreover, the authors of [FO97] have not implemented their method.

The works [VSVA04b, VSVA04a], which appeared concurrently and independently to our results, propose another approach to verification based on inference of regular languages specialised to the verification of properties of systems with FIFO channels. These works use different inference algorithms than the one used in our work (they use either the RPNI [OG92, Lan92] or the Angluin's L^* algorithm [Ang87]) and apply them to sequences of configurations that may appear in the state space of the examined systems rather than to individual configurations. Recently, the latter method has been generalised [VSVA05] to handle more than only FIFO channel systems and also to handle omega-regular properties. The omega-regular model checking is implemented via learning fixpoints of special functionals over paths in state spaces of the examined systems designed for the particular temporal operators. Moreover, in

[8] There has only recently appeared a result [Ler06] giving a specialised termination guarantee for computing the so-called accelerations [BF04] within an NDD-based symbolic verification of counter systems using a modification of the abstraction based on finite-length languages.

[VV05], an extension to branching-time properties of parameterised networks of processes and of systems with counters has been considered.

Finally, we can see the recent growing interest in applying the inference methods in verification even outside the regular model checking framework. In [CCST05, AMN05], they were, for instance, applied in the framework of assume-guarantee reasoning for inferring the assumptions to be used. In [LRS05], a method based on inference of predicates was used within shape analysis of programs with dynamic data structures making more automated the framework [SRW02] based on 3-valued predicate logic with transitive closure. In [GLP06], inference of regular languages was used to obtain invariants of parameterised networks of processes.

Below, we first describe the Trakhtenbrot-Barzdin algorithm for inference of regular languages and then propose a way to use it for regular model checking. Subsequently, we discuss the experiments we have done (including some heuristics that we have tried as an alternative to the Trakhtenbrot-Barzdin algorithm).

3.6.1 Inference of Regular Languages from Complete Training Sets

Inference of regular languages is a very active research area (see, e.g., [TB73, OG92, Lan92, Dup96]). Basically, the problem is to infer a language from some of its words (and/or words known not to belong to the language). An important notion used in the proposed algorithms is that of a training set (or sample). A *training set* $T = (T^+, T^-)$ is a pair of two disjoint sets $T^+, T^- \subseteq \Sigma^*$ where T^+ contains positive examples (words in the language to be inferred), and T^- contains negative ones. A training set $T = (T^+, T^-)$ is called *n-complete* if $T^+ \cup T^- = \Sigma^{\leq n}$ where $\Sigma^{\leq n} = \{w \in \Sigma^* \mid |w| \leq n\}$.

The notion of n-completeness is crucial for the *Trakhtenbrot-Barzdin algorithm* (TB algorithm for short) [TB73] that we have chosen to be applied in our work out of the existing algorithms for inference of regular languages. We show below that n-complete training sets may easily be obtained in regular model checking.

For $n \in \mathbb{N}$ and a language L, let $L^{\leq n}$ denote the subset of L of words of length at most n, i.e., $L^{\leq n} = \{w \in L \mid |w| \leq n\}$. For a length-preserving transducer τ encoding a one-step transition relation (such that $\iota \subseteq \tau$ for the identity relation ι), a regular set of initial configurations I (encoded via a finite-state automaton *Init*), and a regular set of bad configurations B (represented by a finite-state automaton *Bad*), the regular model checking problem of checking whether $\tau^*(I) \cap B = \emptyset$ holds can be seen as a language inference problem in the following way: We want to compute (or at least approximate) the set $\tau^*(I)$. Since τ is length-preserving, the set $\tau^*(I^{\leq n})$ is finite for each n and can be calculated by finitely iterating τ (recall that $\iota \subseteq \tau$). Furthermore, each word of length smaller or equal to n which is not in $\tau^*(I^{\leq n})$ cannot be in $\tau^*(I)$ either. Therefore, the sets $\tau^*(I^{\leq n})$ and $\Sigma^{\leq n} \setminus \tau^*(I^{\leq n})$ can be seen as sets of positive and negative examples of the language $\tau^*(I)$ that

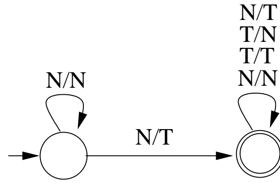


Fig. 3.12. A transducer τ_{Bad} encoding an undesirable behaviour for the token passing example from Fig. 3.1

we want to infer. To be more precise, they contain exactly *all* positive and negative examples of words of the given language up to some length and thus form an n-complete training set.

For increasing n, we get more and more positive and negative examples of the language $\tau^*(I)$—the training set is growing. Therefore, if $\tau^*(I)$ is regular, we will eventually (we do not know when, of course) obtain a training set T big enough in terms of [TB73], and the TB algorithm will allow us to infer $\tau^*(I)$ from T. If $\tau^*(I)$ is not regular, it is still possible to perhaps get an overapproximation sufficient to prove the property of interest.

The same approach can also be used to try to infer the relation τ^* for length-preserving transducers by considering as the alphabet pairs of letters and by computing τ^* restricted to words of length smaller or equal to n (and then generalising it according to the TB algorithm). In this case, we may specify the undesirable property of the system using a length-preserving transducer τ_{Bad} and consider the verification problem of checking whether $\tau^* \cap \tau_{Bad} = \emptyset$. An example of a transducer τ_{Bad} specifying an undesirable behaviour for the simple token passing protocol from Fig. 3.1 is shown in Fig. 3.12—it states that the token should never be passed to the left (nor a new token should be created).

Before going into more details of the technique, let us comment a bit more on why dealing with length-preserving transducers that we now concentrate on is not as restrictive as it might seem. As we have already indicated, this is because for finite runs that suffice for verification of safety properties, we can always replace adding and removing of symbols by rewriting special blank symbols $\perp$ added in advance into the initial configurations. More precisely, we can add self-loops labelled with $\perp/\perp$ to every state of τ, replace every ε/a by a $\perp/a$ transition, every a/ε transition by an $a/\perp$ one, and add $\perp$-labelled self-loops at every state of the automata $Init$ and Bad (provided they are used). Moreover, we later show an extension of our technique to non-length-preserving transducers too.

The Trakhtenbrot-Barzdin Algorithm

To describe the TB algorithm and its use in regular model checking in detail, we need some more definitions. A deterministic finite-state automaton A is

called *consistent* with a training set $T = (T^+, T^-)$ if $T^+ \subseteq L(A)$ and $L(A) \cap T^- = \emptyset$. A training set $T = (T^+, T^-)$ is *n-complete wrt. a deterministic finite-state automaton A* if $T^+ = L^{\leq n}(A)$. Given an n-complete training set $T = (T^+, T^-)$, we call a deterministic finite automaton $A_T = (Q, \Sigma, \delta, q_0, F)$ the *prefix-tree automaton* of T if $L(A) = T^+$, A_T has the form of a tree and does not contain any nodes with the empty language (provided $T^+ \neq \emptyset$).

Furthermore, we define the *depth of an automaton* $M = (Q, \Sigma, \delta, q_0, F)$ denoted d_M as the maximum length of the shortest paths leading to the particular states of M from the initial state, i.e., $d_M = max_{q \in Q}\ min_{w \in \Sigma^* \wedge q_0 \xrightarrow{w} q} |w|$. We say that two states $q, q' \in Q$ of M are *k-indistinguishable*, which we denote by $q \equiv_k q'$, if $L^{\leq k}(M, q) = L^{\leq k}(M, q')$. We then define the *degree of distinguishability* ρ_M of M as the minimal k such that any two states q, q' of M are k-distinguishable, i.e., $q \not\equiv_k q'$.

We now describe a slightly modified version of the Trakhtenbrot-Barzdin algorithm which computes an inferred deterministic finite-state automaton (also called *target automaton*) with the minimal number of states consistent with a given n-complete training set. Let $T = (T^+, T^-)$ be an n-complete training set and A_T the deterministic prefix-tree automaton of T. Obviously, states of A_T must correspond to states of the target automaton $\bar{A}_T$ since it must accept all words accepted by A_T. Several different states of A_T can, however, correspond to the same state of $\bar{A}_T$. Hence, the basic idea of the algorithm is to collapse two states of A_T if this does not lead to a word of length shorter or equal to n being accepted though it is not accepted by A_T. Two states q and q' in A_T can be safely collapsed if they are *compatible*, i.e., if they are k-indistinguishable ($q \equiv_k q'$) where k is the minimum of the heights of the subtrees starting at q and q'.

Let *succ* be the function which associates to each state in A_T the successor in a breadth-first ordering. The algorithm modifies A_T by identifying compatible states:

Algorithm 3.6.1

```
input: A_T with initial state q0
q1 := q0;
while there is a successor of q1 in A_T do
  q1 := succ(q1);
  q2 := q0;
  while q1 ≠ q2 and not compatible(q1, q2) do
    q2 := succ(q2);
  od
  if q1 ≠ q2 and compatible(q1, q2) then
     let the transition from father(q1) to q1 point to q2 and
     erase q1 and all its children from A_T;
od
output: the modified automaton A_T
```

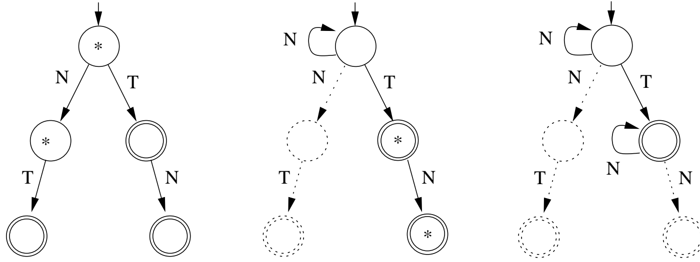


Fig. 3.13. A 2-complete training set and the different stages of the TB algorithm

Notice that the original algorithm [TB73] expects on its input trees which are complete (each internal node has a son for each letter). In our setting, this is not necessary since we only consider languages and not the output behaviour of automata. The algorithm has a complexity of $O(mn^2)$ where m is the size of A_T and n the size of the target automaton. In Fig. 3.13 and 3.14, we give the different stages of the TB algorithm run on a 2-complete training set for $\tau^*(I)$ and a 3-complete set for τ^* of the simple token passing protocol from Fig. 3.1 (with the identity relation added into τ). Two collapsed compatible states at each stage are marked with $*$. Notice that in the first case, $\tau^*(I)$ is obtained exactly as the result of the algorithm, whereas in the other case, the result is an overapproximation of τ^*.

We have the following theorem [TB73].

Theorem 3.6.1 *Let T be an n-complete training set. Algorithm 3.6.1 computes a deterministic finite-state automaton $\bar{A}_T$ with a minimal number of states consistent with T.*

Notice that there could be several different deterministic finite-state automata with a minimal number of states consistent with T—the output of the TB algorithm is just one of them. If all the words of the training set come from some minimal deterministic automaton A, then the TB algorithm is guaranteed to infer it from an n-complete training set if n is sufficiently big with respect to the structure of the automaton. The *degree of reconstructability* r of an automaton A is defined as $r = d + \rho + 1$ where d is the depth and ρ the degree of distinguishability. Then we have the following theorem [TB73].

Theorem 3.6.2 *Given a minimal deterministic finite-state automaton A with degree of reconstructability r and a training set T r-complete wrt. A, Algorithm 3.6.1 computes A (up to isomorphism).*

If A has n states, then in the worst case, $d = \rho = n - 1$ and $r = 2n - 1$. Therefore, the complete training set must contain exponentially (in n) many words. Fortunately, on average [TB73, Lan92], r is much smaller because it

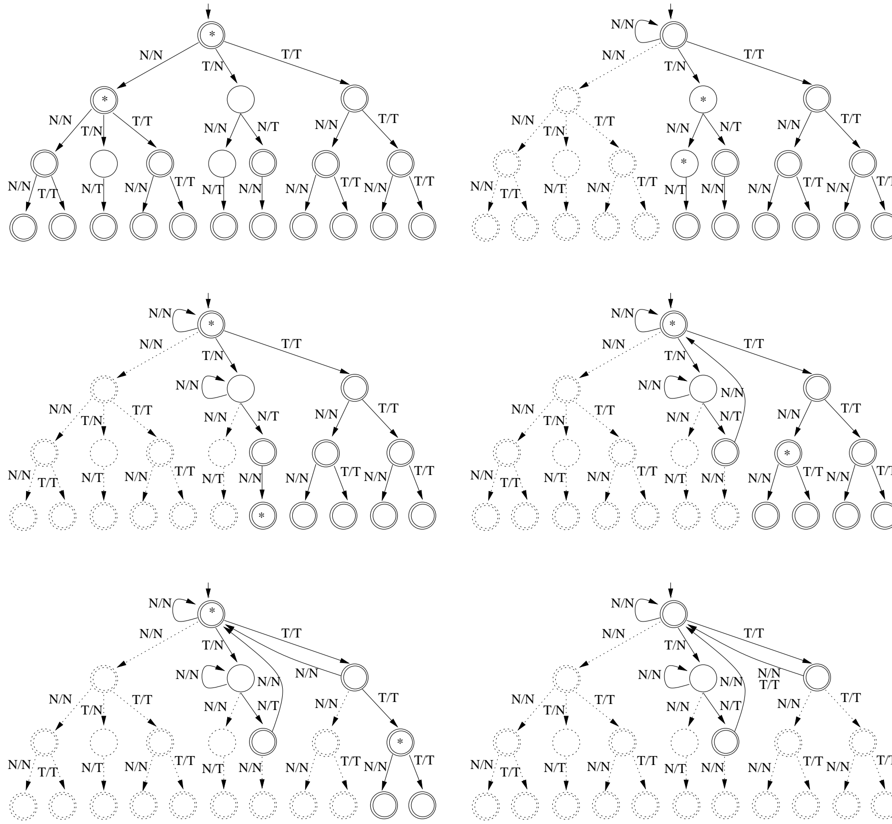


Fig. 3.14. A 3-complete training set and the different stages of the TB algorithm

can be shown that the average value of d is $Clog_{|\Sigma|}(n)$ where C is a constant depending on Σ and $\rho = log_{|\Sigma|}log_2(n)$. This means that on average, the degree of reconstructability r is small compared to the size of the automaton and only small complete training sets (polynomial in n) are needed to reconstruct it.

3.6.2 The Model Checking Algorithm

In this section, we describe our model checking algorithm based on inference of regular languages. We start with a basic version of the algorithm and then present a few modifications and extensions to this algorithm.

The Basic Model Checking Algorithm

The idea of our algorithm is to compute bigger and bigger complete training sets coming from the language $\tau^*(I)$, infer an automaton from them, and

test whether this inferred automaton is an invariant sufficient to prove the property. As the inference algorithm, one can—in principle—use any inference algorithm based on positive and negative examples proposed in the literature (as, e.g., RPNI [OG92, Lan92]). In our work, we use the Trakhtenbrot-Barzdin algorithm discussed above because it works with a complete training set and is guaranteed to output the original automaton if given sufficiently big training sets. Below, we, however, describe our model checking algorithm in a general way.

Algorithm 3.6.2

`input`: *a length-preserving transducer τ, a regular set of initial configurations I, and a regular set of bad configurations B*
$i := 1$; `/*` *i can be initialised differently too.* `*/`
`repeat`
  $C := \tau^*(I^{\leq i})$;
  $\overline{C} := \Sigma^{\leq i} \setminus C$;
  `if` $B \cap C \neq \emptyset$ `then`
    `output:` *property violated;*
  $A := inference(C, \overline{C})$;
  $i := i + 1$;
`until` $\tau(L(A)) \subseteq L(A)$ `and` $I \subseteq L(A)$ `and` $L(A) \cap B = \emptyset$;
`output:` *property satisfied*

When we use the version of the TB algorithm described as Algorithm 3.6.1 for inference in Algorithm 3.6.2, the call of $inference(C, \overline{C})$ invokes Algorithm 3.6.1 with the prefix-tree automaton of C as input. Then, the computation of $\overline{C}$ is not necessary.

In the simple token passing example, to verify the property $\tau^*(I) \cap B = \emptyset$, the algorithm stops for $i = 2$, and the inferred invariant is exactly $\tau^*(I)$. Moreover, if we specify the undesirable property for our example using the transducer τ_{Bad} shown in Fig. 3.12 and we check whether $\tau^* \cap \tau_{Bad} = \emptyset$, the algorithm stops for $i = 3$ with the overapproximation of τ^* shown in Fig. 3.14.

Notice that to calculate $\tau^*(I^{\leq i})$, one can reuse $\tau^*(I^{\leq i-1})$ and only calculate the reachable configurations of size i. The algorithm tries bigger and bigger training sets until it terminates because it either finds a counterexample to the property of interest, or an invariant including the initial states and not intersecting the "bad" set B. The test $I \subseteq L(A)$ is necessary because for small i, the examples generated may not suffice to reconstruct I. An alternative way would be to set the initial value of i wrt. I, but according to our experience, this is not always the best choice.

If $\tau^*(I)$ is regular, the algorithm always terminates.

Theorem 3.6.3 *Let τ be a length-preserving transducer and I and B two regular sets. If $\tau^*(I)$ is regular, then Algorithm 3.6.2 with Algorithm 3.6.1 used as the inference algorithm always terminates.*

Proof. If $\tau^*(I) \cap B \neq \emptyset$, then there exists a word $w \in \tau^*(I) \cap B$ of some size n. Since τ is length-preserving, $w \in \tau^*(I^{\leq n})$ and the algorithm terminates after at most n iterations. If $\tau^*(I) \cap B = \emptyset$, then because of Theorem 3.6.2, the algorithm stops after at most r (the degree of reconstructability of $\tau^*(I)$) steps. □

Notice that termination of the algorithm with the property verified means that an invariant precise enough to prove the property was inferred. In general, we cannot check whether we have inferred the exact reachability set $\tau^*(I)$. This is clear, e.g., from the fact that for lossy channel systems, $\tau^*(I)$ is known to be regular [CFI96a, ABJ98] but not computable [ABB01, May00b]. From Theorem 3.6.3, we get easily the following.

Corollary 3.6.1 *The regular model checking problem of checking whether $\tau^*(I) \cap B = \emptyset$ is decidable for given regular sets I and B and a length-preserving finite-state transducer τ if $\tau^*(I)$ is regular.*

The above is not very surprising as we can give two semi-decision procedures for the problem: one looking for bigger and bigger counterexamples, the other one enumerating all regular languages and checking for invariants (as explained in [Pac87] for FIFO-channel systems). Our algorithm provides a clever way to enumerate regular languages being candidates for an invariant.

Finally, without going into detail, it is clear that in a very similar way as above, we can deal with transition relations too. Let τ_{Bad} be a finite-state length-preserving transducer describing reachability relations among configurations that are not allowed. Then, we have the following result.

Corollary 3.6.2 *The regular model checking problem of checking whether $\tau^* \cap \tau_{Bad} = \emptyset$ is decidable for given length-preserving finite-state transducers τ and τ_{Bad} if τ^* is regular.*

A Few Modifications and Extensions of the Basic Algorithm

Algorithm 3.6.2 can easily be modified to handle non-length-preserving transducers τ too: When we calculate the fixpoint $\tau^*(I)$, after each step, we always intersect the reachable configurations with $\Sigma^{\leq n}$. In this way, the fixpoint computation will always terminate. However, the training set is not guaranteed to be complete anymore as it is possible that some configuration from $\Sigma^{\leq n}$ is reachable only via configurations of length longer than n. Therefore, termination of the model-checking algorithm is not in general insured even for

regular $\tau^*(I)$. However, according to our practical experience, the method still behaves well for various concrete examples.

Further, instead of running Algorithm 3.6.1, on which Algorithm 3.6.2 is based, over a prefix-tree automaton, let us note that it may be directly run over the minimum deterministic automaton that is often in practice the result of computing $C := \tau^*(I^{\leq i})$. Because it does not contain any loops, such a minimum deterministic automaton has the form of a DAG. Working with a prefix-tree automaton can be emulated over the DAG by remembering the depth i of the tree and the number of steps that were taken to get to the node q_1. The difference of these two values may be used to deduce the length of words whose acceptance from q_1 and q_2 should be considered in Algorithm 3.6.1. This step is necessary because several states of the prefix-tree automaton with different depths (corresponding to different incoming paths) may be merged into a single DAG state, and the length to be considered cannot be deduced just from the state itself.

3.6.3 Experiments with the Inference-based RMC

We have implemented the ideas discussed above in a prototype tool written in YAP Prolog using the FSA library [vN04]. We applied the tool to a variety of different verification tasks described below. In addition, we have then also replaced the TB inference algorithm in our model checking schema by several methods inspired by those described in [BHV04] (Section 3.5) as we briefly report at the end of the section.

The Experiments Done and the Results Obtained Using the TB Algorithm

The experiments were similar to those used in [BHV04] and presented in more detail in Section 3.5.5. They included verification of *parametric systems* (in the form of a bit idealised parametric Bakery, Burns, Dijkstra, and Szymanski algorithms of mutual exclusion), a simple *push-down system* modelling a program with several mutually recursive procedures (the plotter control example from [EHRS00]), the alternating bit protocol as a representative of *systems with (lossy) queues*, a Petri net modelling the readers-writers problem with dynamically arising and disappearing processes that can be considered an example of a system with *unbounded counters* (whose values were encoded in parallel in unary), and a procedure for reversing linear lists as a representative of systems with *unbounded recursive data structures*. The simplified Bakery mutual exclusion algorithm was modelled in several ways: with a parametric number of processes and the values of tickets encoded by the positions of the appropriate processes in the word representing a configuration, and with a bounded number of processes (three to five) with the tickets modelled by explicit counters with values encoded in parallel either in binary (as in NDDs) or in unary.

As in Section 3.5.5, we have mostly considered verification of *invariance properties* that can be directly handled using reachability verification. However, we have tried dealing with some more complex properties too. The push-down example where we checked some constraint on the calling order of the procedures is an example of dealing with a bit *more complex safety properties*—to transform it to a reachability problem, we manually composed the appropriate safety automaton with the model of the system. We have also verified *communal liveness* in the Bakery example. In this case, we have manually composed the appropriate Büchi automaton with the system being verified. We have mostly considered *correct systems*, but we have as well run the tool over a *faulty version* of one of the considered systems—namely the Readers/Writers example where we omitted one of the Petri net arcs. We have mostly worked on the level of dealing with *reachability sets*, but in the example of reversing lists, we have also worked with a *reachability relation* represented by a transducer. (Using the reachability set computation, we checked that the procedure outputs a list, but using the reachability relation—restricted to reachability from initial states, i.e., to $\iota_I \circ \tau^*$, we checked that the output list is a reversion of the input one.) Finally, in the experiments, we were trying both *forward* and *backward verification*—i.e., starting from the initial states or the "bad" states.

The results of the experiments obtained on a computer based on an Intel Pentium 4 processor at 1.7 GHz are summarised in Table 3.3. For each experiment, we give the best result obtained. We say whether it was within forward or backward verification and what the initial length of the words in the sample was (the considered values were: 1, $|Q_{Bad}|$, $2|Q_{Bad}|$, $|Q_{Bad}|/2$, $|Q_{Init}|$, $2|Q_{Init}|$, and $|Q_{Init}|/2$). When we compare these results with those of [BHV04] (which belong among the best in the field), we see that they are usually a bit slower but comparable. In one case (the ABP example), the inference method was even faster than the one of [BHV04]. Such results are very positive taking into account that termination guarantees for [BHV04] are not very clear.

Finally, in the r_g columns of Table 3.3, we give the percentage of time spent in generating the finite sample, which indicates that the treatment of this part of our method deserves a special attention in the future optimisations.

Using Other Inference Methods than the TB Algorithm

In a series of additional experiments, we have then replaced the use of the TB algorithm in our model checking schema by several heuristics inspired by [BHV04]. In particular, we have tried to generalise the obtained samples represented by finite automata by collapsing *any* states of these automata having the same forward (or backward) languages of words up to a certain length (wrt. the given set of final states or, alternatively, considering all states to be final). Although the use of these heuristics turned out to be mostly slower than the use of the TB algorithm, there were also cases (e.g., the Bakery

Table 3.3. Some results of experimenting with verification based on inference of regular languages

Experiment	Best setting	T [sec]	r_g [%]
Bakery	Bw, $\|Q_{Bad}\|$	0.03	50
Bakery communal liveness	Fw, $2\|Q_{Init}\|$	0.36	90
Bakery counters – 3 processes	Bw, $2\|Q_{Bad}\|$	8.69	70
Bakery counters – 4 processes	Fw, $\|Q_{Bad}\|$	143	92
Bakery – 5 processes, unary encoding	Fw, $2\|Q_{Init}\|$	229	45
ABP	Bw, $2\|Q_{Bad}\|$	0.03	50
Burns	Fw, $2\|Q_{Init}\|$	0.77	98
Dijkstra	Fw, $\|Q_{Bad}\|/2$	1.16	92
Push-down System	Bw, $\|Q_{Bad}\|$	0.04	63
Petri net – Readers Writers	Fw, $\|Q_{Bad}\|/2$	323	90
Petri net – Faulty Readers Writers	Fw, $2\|Q_{Init}\|$	1.48	54
Szymanski	Fw, $\|Q_{Bad}\|$	0.76	94
List Reversion	Fw, $\|Q_{Init}\|$	1.64	90
List reversion – trans. relation	Fw, $\|Q_{Init}\|/2$	40.5	69

communal liveness or Burns experiments) where they were up to three times faster.

Inspired by the above, in the future, more experiments based on the generality of our model checking schema, which allows us to plug in various inference algorithms, such as [OG92, Lan92], can be done. An investigation of incremental inference algorithms like RPNI2 [Dup96] could especially be interesting. They are based on refining an inference hypothesis when new positive and negative examples are provided. They could easily be used in our framework since we compute longer and longer configurations. A further optimisation could be a use of dedicated finite-state model checkers to compute the set of reachable configurations of bounded length efficiently.

3.7 Summary and Extensions of RMC

We have presented regular model checking as a generic and usually fully automated method for verifying a broad spectrum of various kinds of infinite-state and/or parameterised systems. In detail, we have discussed two particular approaches to regular model checking, namely abstract regular model checking and one of the existing concepts of regular model checking based on language inference. These approaches constitute our original contribution to the research on regular model checking achieved in a tight cooperation with our foreign partners. Abstract regular model checking offers a very high efficiency

whereas our inference-based approach a nice compromise between efficiency and termination guarantees.

Currently, we are working on a better implementation of the techniques, on their possible optimisations for different contexts, and also on their use in various extensions of regular model checking as mentioned below.

An application domain on which we in particular concentrated in the recent time is the domain of verifying programs with dynamic linked data structures. In [BHMV05], we proposed an original approach of using regular model checking in verification of *programs with dynamic linked data structures* with one next pointer, which covers the frequently used structures of singly-linked lists and cyclic lists. We will discuss this approach in Chapter 4.

The first possible extension of regular model checking that one can think of is dealing with regular languages not of finite words, but of more complex objects. The most natural extension in this direction is probably dealing with *regular sets of finite trees* (i.e., *regular tree languages*) which we discuss in Chapter 5. Such an extension appears to be very useful as trees are quite common in computer science where they constitute the topology of configurations of various distributed (parameterised) protocols, cryptographic protocols, programs with recursion and parallelism, programs with dynamic data structures, etc.

Moreover, on top of a tree skeleton, one can successfully code even *more general finite graph structures*. For this, one can, e.g., use routing expressions [KS93] representing additional links between some tree nodes. The use of such techniques in regular tree model checking is another subject of our current work. A direct use of automata on more general than tree structures remains an open question—first, suitable notions of graph automata are to be properly investigated.

Another extension of regular model checking for dealing with more complex objects than finite words is dealing with infinite words leading to the so-called *omega regular model checking*. Omega regular model checking can be useful in verification of liveness properties [BLW05, VSVA05] and also, e.g., when dealing with systems whose configurations are vectors of real numbers [BLW04]. Real numbers can be encoded in binary but we need infinite words in order not to loose preciseness [BRW98]. This leads to a necessity of dealing with Büchi automata for which, however, some needed automata-theoretic operations are problematic (projection, determinisation, complementation). To circumvent this problem, [BJW01] proposes a use of the so-called weak Büchi automata[9] for which all the needed operations are much easier.

Finally, one can think of going beyond (word/tree/omega) regular model checking by dealing with various classes of *non-regular languages* for which

[9] A weak Büchi automaton is a Büchi automaton whose set of states may be partitioned into exclusively accepting or non-accepting disjoint subsets of states over which one can define a partial order such that transitions can only be done from "bigger" to "smaller" states [MSS86].

one can find suitable automata encoding. Here, one can think of using various kinds of push-down automata, automata with constraints, etc., but one has to be very careful to find a class of languages (automata) over which one can perform all the needed language-theoretic operations and tests (or at least safely and efficiently approximate them). Finding such languages and the associated automata is usually not easy. We discuss this issue and our contribution to it, which consists in proposing and applying an original notion of tree automata with size constraints, in Chapter 6.

4

Regular Model Checking and Programs with Pointers

In this chapter, we discuss one of the most interesting and most important applications of abstract regular model checking—namely, verification of *programs manipulating dynamic linked data structures*. Such programs are in general difficult to write and understand, and so the possibility of their formal verification is highly desirable. Formal verification of such programs is, however, a very difficult task too. Dynamic allocation leads to a necessity of dealing with infinite state spaces. Moreover, the objects to be dealt with are the so-called program *stores*, i.e., the dynamic memory part of program configurations containing dynamically allocated memory cells linked by pointers. Program stores have in general the form of graphs whose shape is difficult to be restricted in advance (we call these graphs *heap graphs* or *shape graphs* in the following). The problem is that the linked data structures may fulfil some shape invariants at certain program points, but these invariants may be temporarily broken in various ways at other program points while the program is performing some operations over the given data structures.

Due to the mentioned usefulness and at the same time complexity, the field of formal verification of linked dynamic data structures is currently a very active research area. A number of different approaches differing in their generality and degree of automation have been published, and new approaches are still being proposed and investigated. We present a short overview of these approaches below. Then, we concentrate on our approach of using abstract regular model checking for automatic verification of sequential non-recursive programs manipulating dynamic linked data structures with one selector. The considered class of structures includes traditional singly-linked lists and circular lists (possibly sharing their parts) that belong among the most commonly used structures in practice. The results are based on [BHMV05] where regular model checking was for the first time systematically used in the given area—before, there has only been an isolated ad-hoc attempt to do so in [BHV04] (mentioned in the previous chapter).

As a part of our contribution, we first provide a *systematic encoding* of the configurations of the considered programs as words over a suitable finite

alphabet. Potentially infinite sets of configurations can then be represented by finite-state automata. Moreover, we propose an *automatic translation* of non-recursive sequential C-like programs (without pointer arithmetics and with suitably abstracted non-pointer data values) into finite-state transducers applicable to the sets of program configurations represented by automata and defining regular relations between these configurations. The translation is done statement-by-statement, and one can then either take a union of all statement transducers or use them separately. Since for some of the pointer manipulating statements, the translation cannot be achieved by providing a direct construction of a single transducer, we propose to simulate them by a repeated application of some auxiliary transducers.

By repeatedly applying the transducer (or transducers) representing a program to the automaton encoding a set of possible initial configurations, one can obtain the sets of configurations reachable in any finite number of steps. It is, however, usually impossible to obtain the set of all reachable configurations in this way—the computation will not stop for most programs with loops. One thus has to consider techniques that will accelerate the computation achieving termination as often as possible—a general termination result cannot be obtained as the verification problem considered is clearly undecidable.

To accelerate the reachability computation, we use our *abstract regular model checking* framework. However, compared to the results introduced in [BHV04] and presented in Section 3.5, we propose a new set of abstractions that are more tailored for the given domain and thus promise much better performance results. These techniques are based on *new language abstractions*, which contrary to those introduced in [BHV04], are not defined on the representation structures (i.e., the automata representing sets of configurations), but *defined on words* (corresponding to configurations). Such abstractions are defined by means of finite-state transducers following different generic schemas. The definitions of these abstractions are guided by the observation that in the configurations of the programs we consider there are some repeated patterns for which it is sufficient to remember their number of repetitions precisely up to some fixed bound. If the number of repetitions is higher than the bound, we abstract it to an arbitrary value. The abstraction schemas we define are refinable in the sense that they define infinite sequences of abstraction mappings with increasing precision. Therefore, our verification approach is based on computing abstractions of the sets of reachable configurations, and on refining the abstractions when spurious counterexamples are detected.

These techniques allow us to *fully automatically compute* safe overapproximations of the state space of programs with 1-selector-linked dynamic data structures from whose elements the non-pointer fields are abstracted away. In this way, we can automatically check many important safety properties related to a correct use of dynamically allocated memory—absence of null pointer dereferences, working with uninitialised pointers, memory leakage (i.e., checking that there does not arise any unfreed and unaccessible garbage), etc.

Furthermore, we can automatically handle the cases where a finite number of elements of the considered dynamic data structures are allowed to carry other than pointer fields. Using this fact and a simple technique which we propose for describing the desired input/output configurations, we can then automatically verify various properties relating the input and output of the considered programs (e.g., that the output of a list reversing procedure is really exactly the reverse of the input list, etc.). Finally, we show how the techniques can be applied to dealing with linked dynamic data structures whose elements contain any data fields of finite type too. In this case, a little help from the user may be needed. However, the manual help of the user may be replaced by using a heuristic that we have recently proposed (and which we briefly mention at the end of the chapter), or the user may decide to use some of the slower, fully automatic, general-purpose acceleration methods from Section 3.5.

We have implemented the proposed techniques in a prototype tool and tried it out on a number of procedures manipulating classical singly-linked lists as well as cyclic lists. The results are very encouraging and show the applicability of our approach. Moreover, the approach provides a starting point for using regular tree model checking for handling programs over more complex data structures as was indeed recently done in [BHRV06b]. We will briefly discuss the generalisation based on abstract regular tree model checking at the end of Chapter 5. This approach belongs among the most general and at the same time fully automated approaches proposed.

Below, we first give an overview of the various proposed approaches to verification of programs with dynamic linked data structures—we do not consider only singly-linked structures but also more general structures (which we cannot verify by our original techniques presented in this chapter, but which we can handle via their generalisation mentioned in Chapter 5). Then, we present the way we encode programs over singly-linked dynamic data structures in our framework. We go on by discussing the specialised abstractions we proposed. Finally, we give some experimental results and comment on the possible future work.

4.1 An Overview of the Existing Approaches

There exists a wealth of approaches to the verification of programs with dynamic linked data structures based on various principles and having various advantages and disadvantages. We first mention two often cited approaches linked to the tools PALE and TVLA. Then, we discuss other approaches based on logics, automata as well as other principles. However, none of these approaches can be considered to give the final answer yet, and so—as we have already said above—the research in the given area is very live. That is why,

despite we hope to give a wide and interesting overview of the existing alternative approaches, it is likely that already during writing of these thesis some new approaches could have appeared.

4.1.1 PALE

PALE (i.e., the *Pointer Assertion Logic Engine*) [MS01] allows a *semi-automated* verification of programs manipulating dynamic linked data structures. The semi-automation means that the user of PALE must manually provide *loop invariants*—loop free code fragments can already be handled automatically. The first version of the approach behind PALE [JJKS97] was intended for programs manipulating singly-linked lists, and the second version [EMS00] for programs manipulating trees. In the last version [MS01], PALE handles programs manipulating a quite general class of linked data structures definable using the so-called *graph types* [KS93].

Graph types use a *tree backbone* to encode some of the next pointers involved in the considered linked data structures. The remaining next pointers may be encoded via the so-called *routing expressions* that allow one to go up and down in the tree backbone and also test properties of the encountered nodes (testing whether they are leaves, the root, or they correspond to a certain variant of a node of the given structure). This way singly-linked and doubly linked lists, trees, as well various other structures may be encoded (including, e.g., lists with head/tail pointers, trees with leaves interconnected into a list, trees with nodes additionally linked in the post-order way, etc.). A restriction is that the destination of a routing expression must be deterministic. Moreover, the meaning of next pointers is fixed in advance (though in possibly different ways for different program points).

PALE uses a special *monadic second-order logic on graph types* to express entry and exit conditions of programs, assertions, and loop invariants that must be provided by the user (and that talk about the shape properties of the involved linked data structures). The various pointer manipulating statements are automatically translated to syntactic manipulations on these formulae. Starting from an entry condition or an invariant, PALE can compute a formula expressing the effect of a loop free code fragment. This formula may then combined with the negation of an assertion, loop invariant, or exit condition to check whether some undesirable behaviour exists. For deciding the formulae, an encoding to the *weak monadic second-order theory of two successors* (WS2S) is used.

A decision procedure for WS2S is implemented in MONA [KM01] using the fact that the set of satisfying interpretations of a WS2S formula can be encoded using a finite tree automaton. The decision procedure is non-elementary. To increase the efficiency in as many practical examples as possible, MONA is based on BDDs [Bry86] and a special concept of the so-called *guided-tree automata* [BKR97]. Experiments on multiple practical examples show that

these heuristics are really quite powerful. However, a problem is still the considerable need of human ingenuity in specifying loop invariants that can be sometimes quite tricky.[1]

4.1.2 TVLA

TVLA (the *Three Valued Logic Analyser*) [SRW02] is based on using the Kleene's *3-valued interpretation* [Kle87] of *predicate logic with transitive closure* on graphs. Predicate logic with transitive closure is used for defining the so-called *core* and *instrumentation predicates* over particular memory nodes or their pairs. Core predicates capture the basic semantics of memory structures (e.g., that a node is pointed to by some variable, that a node is a successor of some other node, etc.). Instrumentation predicates are defined over the core predicates and are specific to particular classes of linked data structures—e.g., they can say that a memory node is shared, that a node n fulfils the condition of doubly-linked lists (requiring that when we go forward from n and then immediately backward, we get back to n), etc.

Sets of reachable memory configurations are represented in an abstract way as *3-valued logical structures*—i.e., as a set of memory nodes and an interpretation of the involved predicates over them. A finite representation of infinite sets of memory configurations is achieved via the so-called *canonical abstraction* which introduces *summary nodes*—roughly speaking, the concrete memory nodes that satisfy the same *abstraction predicates* (a designated subset of the involved predicates) are merged into a single summary node. When this abstraction happens, some of the predicates may loose a definite value on some of the nodes. That is why the third logical value interpreted as "may be" is used.

The semantics of particular program statements is encoded using *predicate update formulae* linking the current and future values of the predicates. Predicate update formulae are provided for core predicates; for instrumentation predicates, they are either provided manually, or we can use heuristics to have them derived automatically [RSL03]. To make the method more precise, the update on 3-valued structures first partly concretises the structures via the so-called *focus* operation (a single structure may be split to several more precise ones where the particular summary nodes are replaced by more precisely described nodes allowing one to perform the updates in a definite way). Then, the updates are performed, and finally, the resulting structures are abstracted again. In the process, some basic consistency checks are also performed in the so-called *coerce* operation in order to rule out clearly impossible cases. The focus and coerce operations may in effect also cause the so-called *materialisation* of a non-summary node from a summary one.

[1] We have our experience with such a task from verifying preservation of the full definition of binary search trees after certain operations in [EV05].

The method is quite general, but in the original version described above still not fully automatic as the user has to come up with the right instrumentation predicates (which might not always be easy as we experienced, e.g., in [EV05], or as the work presenting the right predicates for working with circular lists [MYRS05] shows). Recently, a heuristic based on inductive logic programming has been introduced to automatically learn instrumentation predicates [LRS05]. The learning approach has so-far been tested on several examples of sorting algorithms and algorithms over binary search trees. A further investigation of heuristics for learning instrumentation predicates as well as questions of improving the performance and scalability of the approach seem to belong among quite interesting subjects for future work in the given area.

4.1.3 Other Logic-based Approaches

Separation Logic

Separation logic [Rey02] is an extension of Hoare logic for reasoning about programs manipulating pointers and dynamic linked data structures. The key new ingredient in separation logic is the *separating conjunction* $*$. The formula $P * Q$ holds iff P and Q hold in disjoint parts of the memory. The introduction of the separating conjunction allows a local, concise, and modular reasoning about heap manipulations. Otherwise, one has to always reason globally about the entire heap—a modification to a single cell may have a vast impact due to the possible *aliasing* in the heap, i.e., due to different next pointer chains leading to the same cell. Separation logic comes with a number of axioms and inference rules and allows for manual or semi-automated verification (theorem proving). However, recently, there are beginning to appear works on using separation logic (or at least the idea of the separating conjunction) as a symbolic model of memory configurations in fully-automated approaches. The work [DOY06] presents a fully automated method for verification of safety of programs manipulating singly-linked lists using an acceleration of the computation based on (roughly speaking) ignoring all facts that depend on a point in a list that is not a named position. In [DBCO06], verification of termination is even considered, and in [CDOY06], a possibility of dealing with pointer arithmetics is discussed.

Alias Logic

Alias logic [BIL03] is another logic for reasoning about programs manipulating dynamic linked data structures. It allows for explicit reasoning about aliasing. The work [BIL03] presents a proof system for the logic. Let us add that the logic is defined over a *storeless memory model* (i.e., the heap is not represented as a graph) viewing a heap as a collection of languages: a memory node is represented by the regular language of the paths leading to it through the heap interpreted as a finite automaton with the given node taken as the only

final state, and the initial states corresponding to the values of the pointer variables. A similar approach is used in the works of Jonker, Deutsch, and Venet mentioned below.

Predicate Abstraction

A lot of the recent successes of software model checking is due to the technique of *predicate abstraction* [GS97], which abstracts a concrete program to a boolean one where the boolean variables record various predicates about the concrete state of the program (e.g., the fact that the value of a certain integer variable is positive, that the value of a certain integer variable is bigger than that of another integer variable, etc.). However, the traditional works on software model checking using predicate abstraction do not consider programs manipulating dynamic linked data structures.

An attempt to use predicate abstraction over dynamic linked data structures was published in [DN03]. The approach concentrates on using various *reachability predicates* and from them derived predicates like sharing and cyclicity. The method starts with the predicates used in defining the shape properties to be checked and uses a specially proposed *weakest precondition computation* to discover further predicates to be tracked at particular program points. A human interaction is needed to generalise the obtained predicates such that they are sufficient for verification of the given properties. The method has been tested on several classical programs manipulating singly-linked lists.

Another work applying predicate abstraction to verification of programs manipulating dynamic linked data structures is [BPZ05]. In particular, singly-linked data structures are considered. The work uses predicates recording the fact that *some variable is null* and that from some variable the node pointed to by some other variable is *transitively reachable*. The initial predicates are taken from the program conditions and from the condition to be verified. If more predicates are needed, they are to be added manually. The abstract program is derived automatically through a small model property (cut off) of the logic in which the considered programs are specified. The work considers even verification of termination via manually provided ranking functions.

Predicate abstraction over programs manipulating possibly cyclic singly-linked lists is discussed also in [MYRS05], which considers the use of canonical abstraction (the abstraction behind TVLA) too. The predicates used record *aliasing between pointer variables* and the *existence of uninterrupted list segments between pointer variables* (or a pointer variable and null) of length one, two, or longer—uninterrupted segments are segments into the middle of which no pointer variable is pointing and no sharing occurs there. The best possible predicate transformer is provided for these predicates and the common pointer manipulating statements. The corresponding canonical abstraction uses the predicates proposed in [SRW02] and, in addition, predicates recording the existence of uninterrupted list segments between pointer variables (of any length

in this case). Although it is not the case of the corresponding predicate and canonical abstraction presented in the work, [MYRS05] also notes that, in general, exponentially more predicates may be needed in predicate abstraction compared to canonical abstraction.

Finally, in [BHT05], the authors do not introduce a new predicate abstraction over linked data structures, but combine *predicate abstraction over non-pointer data structures* of a program with the use of *canonical abstraction over linked data structures*, thus combining TVLA with the software model checker Blast [HJMS03]. The proposed counterexample-guided refinement loop allows to incrementally increase the precision of the employed canonical abstraction. This is achieved by incrementally increasing the set of the tracked pointer variables for which the corresponding points-to predicates are generated, the subset of these variables for which the points-to predicates are considered abstraction predicates in canonical abstraction, and the set of unary node predicates about the contents of memory nodes. Moreover, the integration happens in the context of lazy, interpolant-based analysis, and so not always the entire symbolic state graph must re-generated, and the refinement may be applied to only certain program locations.

First-order Reasoning

There have also been proposed several approaches based on first-order axiomatisations of various notions of reachability in linked structures usable for automated deduction (theorem proving). A recent work is, e.g., [LQ06] that considers singly-linked as well as doubly-linked acyclic as well as cyclic lists. Cyclic lists are supposed to be well-founded in the sense that each loop contains a distinguished *head cell* pointed to by some variable providing a handle on the loop (this condition is checked during the verification). The reachability predicates used in the work capture reachability from a cell without passing a head cell and the notion of the closest head cell. The proposed first-order axiomatisation of these predicates allows for an almost automatic (and sometimes even fully automatic) verification of various practical pointer manipulating procedures (e.g., list reversion, insertion, deletion, union of sets implemented as lists). Moreover, methods based on indexed predicate abstraction can sometimes be used for an automatic synthesis of loop invariants. One of the nice features of this approach is that it can be easily combined with reasoning about, e.g., integers or arrays.

4.1.4 Automata-based Approaches

Automata-based approaches to verification of programs with dynamic linked data structures include, of course, our approach based on *abstract regular model checking* presented in detail below (and its generalisation to *abstract regular tree model checking* mentioned in Chapter 5). In Chapter 6, we further propose an original class of *tree automata with size constraints* applicable in

a semi-automated verification of programs manipulating *balanced* tree structures. Automata are also in the background of some of the above presented logic-based approaches where they may serve as a basis of the corresponding decision procedures as, e.g., in PALE.

Automata in May-alias Analysis

Apart from the above, automata are used, for instance, in [Jon81, Deu94, Ven99] as well. In [Ven99] following the earlier works [Jon81, Deu94], the special problem of *may-alias analysis* is primarily considered. The approach uses a symbolic representation of memory structures consisting of tuples of automata (one for each pointer variable) and alias relations (using constraints). A special form of widening on this representation is used to accelerate the computation. Alias analysis is a bit less ambitious than the other approaches considered here as it concentrates mainly on discovering which pointers (and pointer sequences) may point to the same memory locations at different times.

Top-down Parity Tree Automata

An interesting automata-based approach has recently been presented in [DEG06]. It is based on *top-down parity tree automata* working on memory shape graphs unfolded in a natural way into infinite trees. The authors define a significant fragment of the programming constructions commonly used for dealing with pointers and a fragment of correctness properties yielding a verification problem that is decidable using their approach. In particular, the procedures they consider must have a single pointer variable called a *cursor* that is used for iterating though the structure (with possibly more next pointers) and for modifying the structure (the modifications can in general happen in a bounded neighbourhood of the cursor). Furthermore, the procedures are limited to a *finite number of destructive passes* though the structure. Such procedures can be automatically translated to the considered tree automata that encode in a single merged tree the relation of the memory configuration before and after performing the procedure (the memory nodes in the tree are flagged as nodes newly created, deleted, and preserved). Then, a product of the procedure automaton with an automaton describing possible inputs and with an automaton describing undesirable outputs is created and tested for emptiness. The properties that can be encoded this way include connectivity properties (reachability, cyclicity, sharing), data-dependent properties (sortedness) as well as the basic memory consistency properties (dangling pointers, null pointer dereferences, etc.). The method is polynomial in the size of the checked procedure and the automata describing possible inputs and undesirable outputs.

A variant of the method based on checking satisfiability of formulae of CTL with past temporal operators used for describing the inputs, outputs, and the checked procedures is then also described.

Counter Automata

Finally, we participated on a proposal of a method for verifying programs manipulating singly-linked structures that is based on encoding their operation on given input structures in the form of a *counter automaton* [BBH+06b, BBH+06a] (a work going in a similar direction was reported in [BFL06], and some preliminary work appeared in [BAN04, BI05]). The control states of the counter automaton encode the current shape graph of the memory with all the uninterrupted next pointer chains (i.e., chains with no pointer variables pointing into their middle and with no sharing within them) contracted to a single node. The counters then encode the length of these chains. Program statements are automatically translated to appropriately changing the control state of the counter automaton (i.e., the shape of the memory) as well as the values of the counters. If the program has n pointer variables, we suffice with a counter automaton with at most $2n$ counters and $O((2n)^{2n})$ control states (in practice, these numbers may often be much smaller). The counter automaton precisely encodes the behaviour of the program—in [BBH+06b, BBH+06a], bisimilarity between the operation of the program and the counter automaton is shown.

Most verification problems on the obtained counter automata are of course undecidable, but one can use any of the numerous *semi-algorithms proposed for analysing counter automata*—including, e.g., (abstract) regular model checking—to successfully verify many real-life situations. Moreover, tools like ARMC [Ryb] based on [PR04a, PR04b, PR05, CPR05] can even be used for an automated *termination checking* on the counter automata encoding the programs. In addition, in [BBH+06b, BBH+06a], it is also proved that if the counter automaton derived from a program is *flat* (i.e., if its control graph has no nested loops), reachability and termination analysis of the program are decidable.

Further, we have also extended the technique to automatically track *ordering properties* among the contents of the memory nodes. The resulting technique allowed us, e.g., to fully automatically verify the basic memory consistency properties, shape preservation, reversedness or sortedness as well as termination for the reversion, bubblesort and insertsort procedures working on singly-linked lists.

Finally, recently [HIRV07], we have come up with an application of counter automata, combined with abstract regular tree model checking, for verification of termination of programs manipulating trees (with tree rotations and addition/deletion of leaves as the only destructive pointer updates).

4.1.5 Other Approaches

Timestamps

An interesting approach to alias analysis has been proposed in [Ven04]. It is based on identifying newly allocated memory objects by *numerical times-*

tamps derived from values of the various numerical variables (as, e.g., loop variables) of the analysed program at the time of an object creation. Methods for analysing programs manipulating numerical data are then used to handle relations among the values of the timestamps of the objects that are manipulated at various positions of a program.

Grammar-based Shape Analysis

In [LYY05], a method based on using *context-free grammars with a single attribute in combination with contracted shape graphs* has been proposed. A memory configuration is represented as a memory shape graph whose parts that are not directly pointed by pointer variables and are unshared are contracted to a single summary node with an attached nonterminal. The derivation tree of the non-terminal describes the structure hidden behind the summary node. The grammar rules to be used are automatically derived when contracting (or folding) the shape graphs. Moreover, shape graphs that differ only in the names of their nodes and in the attached non-terminals are unified—the right-hand sides of the rules of the corresponding non-terminals are then merged. Finally, some further widening is done on the grammar rules by unifying two non-terminals (or a non-terminal and a terminal representing null) when they appear in the same position in some right-hand side, and by unifying non-terminals having "similar" right-hand sides (i.e., having right-hand sides of the same length). To encode some form of sharing or loops, the non-terminals in the grammar rules may have an attribute that may refer to a node of a shape graph, and may be passed on in the grammar productions or be eventually used as a terminal symbol.

The involved overapproximation may be quite aggressive (and there is no refinement proposed here), yet the analysis can successfully (and efficiently) be used for verifying many practical procedures manipulating (even quite complex) linked data structures: The method has been tested on sample procedures manipulating singly-linked as well doubly-linked lists, trees—including the Deutsch-Schorr-Waite tree traversal (a non-recursive traversal that temporarily swaps the pointers in a tree to remember the way back to the root), trees with parent pointers, and binomial heaps (based on lists of trees with parent pointers). Structures that cannot be handled are, e.g., doubly-linked lists with additional pointers (such as the tail pointer) where more attributes of the grammar would be needed.

Memory Patterns

In [YKB02], a method for verifying dynamic linked data structures based on abstracting away the number of adjacent occurrences of certain *patterns (subgraphs) in memory shape graphs* was proposed. The memory patterns were to be provided by a user of the method. In [EV05, ČEV06], we have improved the method by a heuristics for an automated discovery of memory patterns. The technique showed to be very successful for programs manipulating dynamic

linked data structures with a *linear skeleton* and possibly some *additional next pointers*. This way, programs manipulating (cyclic as well as acyclic) singly-linked and doubly-linked lists (extended, e.g., with head and/or tail pointers) may be verified. Moreover, recently, a generalisation of the approach to trees was discussed in [ČEV07].

Graph Rewriting

A quite general formalism that can be used (among others) for describing pointer manipulating programs is the formalism of *graph transformation systems* (GTS). In [BCK01], a technique for analysing reachability in GTSs has been proposed. It is based on overapproximating their behaviour by the so-called *Petri graphs*. A Petri graph consists of a hypergraph and a Petri net whose places correspond to the edges of the hypergraph. Reachable markings of the Petri net encode which hyperedges may appear in the encoded hypergraphs. A Petri graph is obtained by unfolding a GTS (starting with the initial graph and extending it gradually by the appropriate encoding of the graphs resulting from applying the rewriting rules) and folding back repeated occurrences of some of the subgraphs resulting from the rewriting. In [BCE+05], this technique has been applied to verification of insertion in red-black trees (not taking into account the balancedness requirement). Encoding general pointer manipulating programs into the framework of GTS and attempts to verify such programs using the techniques of [BCK01] (and their improvements, e.g., by a counterexample guided refinement schema [KK06]) seems to be the future work of the authors of the technique.

Contracted Shape Graphs

In [LAIS06], the authors propose a method intended (in its basic form) for verifying safety properties of programs manipulating dynamic linked data structures that do not contain any cycles even when the orientation of the next pointers is ignored (which includes acyclic singly-linked lists, trees, and trees with a limited amount of sharing). The abstraction on which the method is built is based on *contracting parts of memory heap graphs* that are unshared and do not contain nodes directly pointed by pointer variables into a single summary node. All the next pointers that are originally within the contracted subgraph form self-loops on the appropriate summary node after the abstraction. The work comes with the best abstract transformers defined on the given abstract representation for the various pointer manipulating statements. The method turns out to be quite efficient on a number of practical examples. Moreover, there are extensions to the method that allow one to handle simple loops in memory structures (e.g., cyclic singly-linked lists) and, with a little help from the user, also structures with parent pointers (including doubly-linked lists and trees with parent pointers). Moreover, the restricting conditions may be temporarily broken if they are restored at loop boundaries. Further, a method is provided for computing an abstract overapproximation of

memory configurations satisfying a given formula in the first-order logic with transitive closure. Such conditions may be used by the user for specifying pre- and post-conditions of procedures, and hence a support for a semi-automated modular analysis is obtained.

Boolean Heaps

In [WKZ+06], a complex approach combing a use of (sometimes specifically extended) decision procedures, theorem provers and several new concepts for an abstract representation of heap graphs is proposed. Heaps are represented in the form of the so-called *boolean heaps* that are disjunctions of universally quantified Boolean combinations of unary predicates over heap nodes. In this heap abstraction, the links between heap nodes are hidden into the unary predicates associated with particular heap nodes (as, e.g., the successor of the given heap node is pointed by some pointer variable) rather than being captured by binary predicates. A concrete heap is described by a boolean heap if one of the top-level disjunctions of the boolean heap is a universally quantified formula whose body is satisfied for every node of the heap. In addition, global, nullary predicates about the heap are allowed too (as, e.g., a predicate saying that two pointer variables point to the same memory node). The predicates to be used are partly obtained by a syntactic analysis of the code to be analysed and the provided pre- and post-conditions and structure invariants, and partly provided by the user.

The predicates used in the above may speak about *reachability between memory nodes*. The approach *distinguishes basic and derived next pointers*. For the derived next pointer links, the user must provide invariants defined over the basic links that must hold at procedure entry, exit, and loop points. This information is then used to modify the formulae being dealt such that they speak about a tree skeleton of the structure, and the MONA decision procedure over trees may be used to decide them (other kinds of skeletons may be considered when using other decision procedures). The predicates may further also speak about *other than linking properties*—e.g., they may speak about numerical sortedness of the contents of memory nodes, etc. To deal with them, special methods for *combining decision procedures* and eliminating the quantifiers introduced by the abstraction (by instantiating the quantified nodes wrt. which pointer variables they are pointed to) are proposed.

The approach uses a syntactical weakest precondition computation combined with the use of decision procedures (with the above mentioned extensions) or a theorem prover for entailment checking to derive loop invariants of the procedures being checked. Subsequently, the invariants and the provided pre- and post-conditions are used to generate verification conditions that are then checked by (extended) decision procedures and/or a theorem prover. The approach has been used for verification of certain procedures manipulating singly-linked as well as doubly-linked lists, trees, trees with parent pointers, sorted lists, and two-level skip lists (having an additional level of next pointers skipping some nodes in the list).

4.2 From Programs to Transducers

After having gone through the alternative approaches, we now concentrate in detail on our own approach based on using abstract regular model checking. In this section, we describe the translation we propose for automatic verification of sequential, non-recursive programs with 1-selector-linked dynamic data structures in the framework of regular model checking. Our translation is general enough to cover *any* program of this kind (not containing pointer arithmetics and not explicitly covering the possibly necessary abstraction of non-pointer data).

We first describe how to encode as words program stores (i.e., heap graphs) of the considered class of programs. Then, we propose an encoding of the standard C pointer operations (apart from pointer arithmetics) in the form of transducers. Some of the pointer operations cannot be translated directly to a single transducer, therefore we propose to simulate their effect by computing a limit of a repeated application of certain simple auxiliary transducers.

In the following, we will use as a running example the following procedure reversing a list l. We suppose the data fields normally present in the elements of the data type `List` to be abstracted away and just the next-pointer fields to be preserved.

```
List x,y,l;
l1: x = null;
l2: while (l != null) {  // i.e., if (l!=null) goto l3; else goto l7;
l3:      y = l->next;
l4:      l->next = x;
l5:      x = l;
l6:      l = y; }        // i.e., l = y; goto l2;
l7: l = x;
l8: // end of program
```

Before proceeding to the details of our encoding, let us note that PALE (or more precisely its version for singly-linked structures) based on [JJKS97] uses a similar encoding of configurations as the one we describe in the following. The possibility of sharing parts of the lists is, however, not considered there. Moreover, as we have already said above, there is no translation of the programs to transducers for manipulating sets of configurations in the PALE approach, the effect of the program is expressed by manipulating a logical description of the configurations, and the approach is not as automatic as ours. Furthermore, representations of linked memory structures based on automata were used in [Jon81, Deu94, Ven99, BIL03] too. In [Ven99] following the earlier work of [Jon81, Deu94], the special problem of may-alias analysis is primarily considered and a different symbolic representation of memory structures is used—it is based on tuples of automata (one for each pointer variable) and alias relations (using constraints). In [BIL03], also mentioned above, an alias logic with a Hoare-like proof system is introduced. There, one memory struc-

ture is represented as a collection of automata whereas we represent a set of memory structures with one automaton.

4.2.1 Encoding Stores as Words

Basically, a store is encoded as the concatenation of several words (separated by a special symbol), each of them representing a list of elements. Successive elements of these lists are given from the left to the right, with positions of pointer variables marked by special symbols. We suppose for the moment that list elements contain no data—later we show that adding data of a finite type is not a problem. We also suppose for the beginning that the store does not contain cycles nor shared parts (i.e., no two different next-pointers point to the same list element). To encode such stores as words, we use the following alphabet Σ: For every pointer variable x used in the program at hand, we have $x \in \Sigma$, and Σ further contains the letters | to separate lists (and some special parts of the configurations), / to separate list elements (i.e., / represents a next-pointer), # to express that a next-pointer points to null, and ! to denote that the next-pointer value is undefined.

Then, we can encode a store without sharing and cycles as a sequence of three parts separated by the symbol | as follows:

- The first part contains a sequence of pointer variables whose values are undefined. In order not to have to consider all their possible orderings, we fix in advance a certain ordering on Σ that is respected here as well as in similar situations below.
- The second part contains pointer variables pointing to null.
- Finally, the third part contains the list sequences separated again by the symbol |. Each list sequence is encoded as follows: Every list element is represented by a (possibly empty) sequence of pointer variables pointing to it, lists elements are separated by the symbol /, and lists end either with the symbol # (null) or ! (undefined).

For example, the word $x\ y\ |\ |\ l\ /\ /\ \#\ |$ encodes a possible initial configuration of the list reversion example: x and y are undefined, no variable points to null, and l points to a list with two elements.

Now, regular expressions (or alternatively finite automata) can be used to describe sets of stores. For instance, the regular expression $(x\ y\ |\ |\ l\ /^{+}\ \#\ |) + (x\ y\ |\ l\ |)$ encodes all possible initial stores for our list reversion example.

Notice that in our encoding, we do not allow garbage (parts of the memory not accessible from pointer variables). As soon as an operation creates garbage, an error is reported. In fact, such a situation corresponds to a memory leak in C (in Java, on the other hand, we can always perform “garbage collection” and remove the garbage).

Remark. Clearly, pointer variables appear exactly once in every word. The separator | and the symbols # and ! appear a bounded number of times since

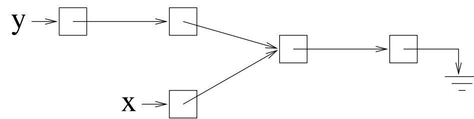


Fig. 4.1. A store with sharing

we do not consider stores with garbage. Finally, the symbol / can appear an unbounded number of times.

4.2.2 Lists with Sharing and/or Loops.

To encode sharing of parts of lists as, for example, in Figure 4.1, we extend the alphabet Σ by a finite set of pairs of markers (m_f, m_t, n_f, n_t, etc.). A "from" marker X_f may be used after a next-pointer sign / to indicate that the given next-pointer points to an element marked by X_t (the corresponding "to" marker). Then, e.g., the word $| \; | \; x \; / \; m_f \; | \; y \; / \; / \; n_f \; | \; n_t \; m_t \; / \; / \; \# \; |$ encodes the store of Figure 4.1.

As one can easily see, the above store could be encoded in several other ways too (for instance, as $| \; | \; x \; / \; n_t \; / \; / \; \# \; | \; y \; / \; / \; n_f \; |$). Although we partially normalise the encoding by imposing a certain ordering on the symbols that are attached to the same memory location, we do not define a canonical representative of the store. However, our experimental results (see Section 4.4) show that this is not an obstacle to a practical applicability of our method. Furthermore, using a canonical form would complicate the encoding of program statements.

Notice also that markers allow us to encode circular lists (as, for example, $| \; | \; x \; n_t \; / \; / \; n_f \; |$ corresponding to a circular list of two elements pointed to by x).

It is not difficult to see that given a store with k pointer variables encoded with more than k pairs of markers, one can encode the same store with at most k markers provided that no garbage is allowed: If a "to" marker is at the beginning of a sequence of cells that is not accessible without using markers, we can put these sequence directly in place of the corresponding "from" marker and save one pair of markers. For example, the store $| \; | \; x \; / \; m_f \; | \; y \; / \; / \; n_f \; | \; n_t \; m_t \; / \; / \; \# \; |$ of Figure 4.1 can be described with one pair of markers as $| \; | \; x \; / \; n_t \; / \; / \; \# \; | \; y \; / \; / \; n_f \; |$ or also as $| \; | \; x \; / \; m_f \; | \; y \; / \; / \; m_t \; / \; / \; \#|$.

Typically, the number of markers that is really needed is even smaller than k as we will demonstrate in our experiments.

4.2.3 Encoding Program Statements as Transducers

We now describe our encoding of program statements as transducers. We consider non-recursive C programs without pointer arithmetics. Initially, we also

suppose all non-pointer data manipulations to be abstracted away—we briefly return to handling them later. Such programs may easily be pre-processed to contain only statements of the form `pointer_assignment; goto l;` or `if (pointer_test) goto l1; else goto l2;`. Moreover, by introducing auxiliary variables, we can eliminate multiple pointer dereferences of the form `x->next->next` and consider single dereferences only.

To encode full configurations of the considered programs, we extend the encoding of stores by adding a letter for the line of the program the control is currently at (followed by a separator |). Moreover, for the needs of our verification procedure, we add a single letter indicating the so-called computation mode. The mode is either n (normal), e (error—a null pointer dereference or working with an undefined pointer has been detected), s (shifting, used later for implementing the pointer manipulation statements that cannot be implemented as a single transducer), and u (unknown result that arises when an insufficient number of markers is used). For instance, the initial configurations of the list reversion example are then $(n\, l_1 \mid x\, y \mid \mid l\, /^+\, \# \mid) + (n\, l_1 \mid x\, y \mid l \mid)$.

Conditional jumps based on tests like `x==null` or `x==y` are now quite easy to encode. The transducer just checks whether `x` is in the null section or in the same section as `y` (taking / and | as section separators), and according to this changes the letter encoding the current line. If `x` or `y` is in the undefined section, we go to the error mode. Similarly, assignments of the form `x=null` or `x=y` are easy to handle—`x` is deleted from its current position (using an x/ε transition) and put to the section of y (using an ε/x transition).

A slightly more involved case is the one of tests based on the `x->next` construct and the one of the `y=x->next` assignment. Apart from generating an error when `x` is undefined or null, one has to consider the successor of `x`, which may involve going from a "from" marker to the appropriate "to" marker. However, as the number of markers is finite, the transducer can easily remember from which marker to which it is going and skip the part of the configuration between these markers.

Adding/Removing Markers

The most difficult case is then the one of the `l->next=x` assignment, which is a destructive pointer update. The transducer first tries to commit the operation by using a pair of unused markers (say m_f/m_t) out of the in advance chosen set of marker pairs (an unused marker pair is one that does not appear in the current configuration word). Then, behind the section of `l`, the transducer puts m_f, and marks the section of `x` by m_t. For instance, in the list reversion procedure, $n\ l_4 \mid \mid \mid x\ /\ /\ \# \mid l\ /\ y\ /\ \# \mid$ is transformed via `l->next=x` into $n\ l_5 \mid \mid \mid m_t\ x\ /\ /\ \# \mid l\ /\ m_f \mid y\ /\ \# \mid$ as shown in Figure 4.2 (a), (b).

However, there may not be any unused markers left. In such a case, the transducer tries to reclaim some by re-arranging the configuration. This can be done by moving some sequence of cells that starts with a "to"

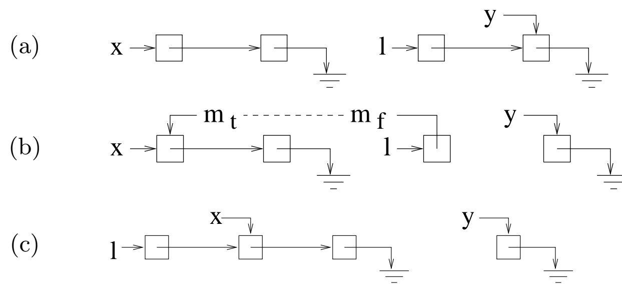


Fig. 4.2. An example store, the store after the statement `l->next=x`, and after a rearrangement

marker directly into the place of the corresponding "from" marker (provided these markers do not constitute a loop). As explained in Section 4.2.2, this is always possible provided the chosen number of pairs of markers is sufficiently big (more than the number of pointer variables). For example, $n\ l_5 \mid \mid \mid m_t\ x\ /\ /\ \#\ \mid l\ /\ m_f\ \mid y\ /\ \#\ \mid$ can be re-arranged to $n\ l_5 \mid \mid \mid l\ /\ x\ /\ /\ \#\ \mid y\ /\ \#\ \mid$ as sketched in Figure 4.2 (c).

The above operation, however, cannot be encoded as a single transducer as it may require an unbounded sequence (such as the list after x in our example) to be shifted to another place, and a finite-state transducer is incapable of remembering such sequences. To circumvent this problem, we use a very simple transducer τ which does one step of the shifting—i.e., it shifts a single element of the sequence by deleting it from its current location and re-producing it at its required location. The desired result is then the limit $\tau^*(Conf)$ where $Conf$ is a regular set of configurations on which the operation is applied. The limit (or an upper approximation of it) is computed using our abstract reachability analysis techniques. In order not to mix half-shifted sequences with the ready-to-use ones, the shifting is done in a special computation mode when no other operations are possible.[2]

If some marker has to be eliminated but this cannot be done, we go to the u mode and stop the computation. Such a situation cannot happen when we use as many markers as pointer variables. Nevertheless, it may happen when the user tries to use a smaller number of them with the aim of reducing the verification time (which is often, but not always possible). If one does not want to use markers at all, the two operations of introducing and eliminating a pair of markers (including shifting) are done at once.

Finally, the remaining `malloc(x)` and `free(x)` operations are again easy to encode. The `malloc(x)` operation introduces a sequence of elements with a single element, pointed to by `x`, and with an undefined successor. The

[2] Shifting could be implemented as an atomic, special purpose (and rather complex) operation directly on the automata too.

`free(x)` operation removes an element, makes `x` and all its aliases undefined, and possibly makes undefined the next-pointer originally leading to `x`.

Adding Data Values to List Elements

The encoding can easily be extended to handle list elements containing data of a finite type. Their values are added into Σ and then every memory cell encoded as a sequence surrounded by / and/or | contains not only the pointers (markers) pointing to it, but also the appropriate data value. The tests and assignments on `*x` may then easily be added by testing whether the appropriate data letter is in the section of `x` or changing the data letter in this section.

4.3 Specialised Abstract Regular Model Checking

We now introduce two specialised classes of abstractions to be used to solve the safety regular model checking problem

$$\tau^*(Init) \cap Bad = \emptyset \tag{4.1}$$

on the above introduced encoding of configurations and transitions of programs manipulating singly linked dynamic data structures.

As described in Section 3.5, the *representation-oriented abstractions* of [BHV04] consist in defining finite-range abstractions on automata used as symbolic representation structures for sets of configurations. The fact that these abstractions are finite-range ensures that a single verification run always terminates leading to either an answer to the given verification question or to a refinement. The general principle of these abstractions is to collapse automata according to some given equivalence relation on their states regardless of the kind of the represented configurations or the analysed system. Here, we adopt an alternative approach by considering *configuration-oriented* abstractions which are defined on configurations. This approach allows us to define abstraction techniques which are more adapted to the application domain we are considering here.

In the next subsections, we discuss two generic schemas for defining families of refinable configuration-oriented abstractions. Instances of these schemas have been implemented in a prototype tool and used in several experiments (see Section 4.4). It turned out that this approach leads to more efficient verification techniques than the previous one for the application domain we are considering here.

4.3.1 Piecewise 0-*k* Counter Abstractions

The idea behind the first configuration-oriented abstraction schema we introduce is to abstract each word by (1) considering some finite decomposition of

it and by (2) applying the 0-k *counter abstraction* to each piece of the word in this decomposition. The 0-k counter abstraction looses the information about the ordering between symbols and only keeps track of their numbers of occurrences up to k. We, in particular, decompose the words in a unique way according to the first occurrence of the particular alphabet symbols.

Formally, given an alphabet Σ and a word $w \in \Sigma^*$, let $dec(w) = (a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$ be such that $w = a_1 w_1 a_2 w_2 \cdots a_n w_n$, $\forall i, j \in \{1, \ldots, n\} : a_i \in \Sigma \wedge (i \neq j \Rightarrow a_i \neq a_j)$, and $\forall i \in \{1, \ldots, n\} : w_i \in \{a_1, \ldots, a_i\}^*$. Given a word w and a symbol a, let $|w|_a$ denote the number of occurrences of a in w. Given $k \in \mathbb{N}^{>0}$, we define a mapping α_k from words to languages such that for every $w \in \Sigma^*$, if $dec(w) = (a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$, then $\alpha_k(w) = a_1 L_1 a_2 L_2 \cdots a_n L_n$ where $\forall i \in \{1, \ldots, n\} : L_i = \{u \in \{a_1, \ldots, a_i\}^* \mid \forall j \in \{1, \ldots, i\} : (|w_i|_{a_j} < k \wedge |u|_{a_j} = |w_i|_{a_j}) \vee (|w_i|_{a_j} \geq k \wedge |u|_{a_j} \geq k)\}$. We generalise α_k from words to languages in the straightforward way in order to obtain a language abstraction. We can easily prove the following proposition.

Proposition 4.3.1 *For every $k \geq 0$, α_k is regular and effectively representable by a finite-state transducer.*

For every given word w, the abstraction α_k applies 0-k counter abstraction to a finite number of subwords of w obtained by a decomposition of w according to the first occurrences of each symbol of the alphabet. Clearly, for every given alphabet Σ, the set of possible 0-k abstractions is finite, and therefore, the number of piecewise 0-k abstractions is also finite since they consist in concatenations of a bounded number of symbols and 0-k abstractions.

Proposition 4.3.2 *For every $k \in \mathbb{N}$, the abstraction α_k is finite-range.*

In fact, below, we consider a generalisation of the above schema obtained as follows. We allow that decompositions may be computed according to the first occurrences of *only a subset* of the alphabet called *decomposition symbols.* Furthermore, we allow that the abstraction does not concern some symbols called *strong symbols* whose all occurrences are preserved at their original positions. Typically, strong symbols are those which are known to have a bounded number of occurrences in all considered words. For instance, in words corresponding to encodings of program configurations, strong symbols correspond to markers, separators, and pointer variables which are known to have either a fixed or a bounded number of occurrences in all configurations.

Formally, let $\Sigma_1, \Sigma_2 \subseteq \Sigma$ be two sets of symbols such that $\Sigma_1 \cap \Sigma_2 = \emptyset$ where Σ_1 is the set of decomposition symbols and Σ_2 is the set of strong symbols. (Notice that there may be symbols which are neither in Σ_1 nor in Σ_2.) Then, given $w \in \Sigma^*$, we define $dec(w)$ to be the decomposition $(a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$ such that (1) $w = a_1 w_1 a_2 w_2 \cdots a_n w_n$, (2) $\forall i \in \{1, \ldots, n\} : a_i \in \Sigma_1 \cup \Sigma_2 \wedge (a_i \in \Sigma_1 \Rightarrow |a_1 a_2 \cdots a_n|_{a_i} = 1)$, and

(3) $\forall i \in \{1, \ldots, n\} : w_i \in (\{a_1, \ldots, a_i\} \setminus \Sigma_2)^*$. Then, for each given k, the abstraction α_k is defined precisely as before.

The previous proposition still holds if the number of occurrences of each strong symbol is bounded. Let us call a p-Σ_2-bounded language any set of words L such that $\forall w \in L : \forall a \ \in \Sigma_2.\ |w|_a \leq p$.

Proposition 4.3.3 *For every bound $p \geq 0$, and for every $k \in \mathbb{N}$, the abstraction α_k is finite-range when it is applied to p-Σ_2-bounded languages.*

As for the abstraction refinement issue, it is easy to see that the abstraction schema introduced above defines a family of refinable abstractions.

Proposition 4.3.4 *For every p-Σ_2-bounded language L, and for every $k \geq 0$, we have $\alpha_{k+1}(L) \subseteq \alpha_k(L)$. Moreover, if L is infinite, then $\alpha_{k+1}(L) \subsetneq \alpha_k(L)$.*

4.3.2 Closure Abstractions

We introduce hereafter another family of regular abstractions. Now, the idea is to apply iteratively extrapolation rules which may be seen as rewriting rules that replace words of the form u^k, for some given word u and a positive integer k, by the language $u^k u^*$.

Let $u \in \Sigma^*$ and let $k \in \mathbb{N}^{>0}$ be a strictly positive integer. A relation $R \subseteq \Sigma^* \times \Sigma^*$ is an *extrapolation rule* wrt. the pair (u, k) if $R = \{(w, w') \in \Sigma^* \times \Sigma^* \mid w = u_1 u^k u_2 \wedge w' \in u_1 u^k u^* u_2\}$. An *extrapolation system* is a finite union of extrapolation rules.

Clearly, for every language L, we have $L \subseteq R(L)$, i.e., R defines a language abstraction. Intuitively, the effect such an abstraction is, whenever there is a word in $w \in L$ which contains at least k successive occurrences of some given word u, to add all words obtained from w by replacing u^k with any word of the form u^p for some $p \geq k$. In fact, we are interested in abstractions which are the result of *iterating extrapolation systems*. Therefore, let us define a *closure abstraction* as the reflexive-transitive closure R^* of some extrapolation system R.

It is easy to see that every extrapolation system corresponds to a regular relation (i.e., a relation definable by a finite-state transducer). The question is whether closure abstractions of regular languages are still regular and effectively computable. In the general case, the answer is not known. However, we provide a reasonable condition on extrapolation systems which guarantees the effective regularity of closure abstractions.

First of all, we can prove that if we consider a single extrapolation rule, the corresponding closure abstraction if effectively computable.

Lemma 4.3.1 *For every extrapolation rule R and for every regular language L, the set $R^*(L)$ is regular and effectively constructible.*

Proof. Let A be an automaton recognising L. Let B be an automaton recognising $u^k u^*$, and let q_i (resp. q_f) be its initial (resp. final) state. Then, for every pair of states (q, q') of A that are related by u^k, we extend A by a unique copy of B and two ϵ transitions $q \xrightarrow{\epsilon} q_i$ and $q_f \xrightarrow{\epsilon} q'$ (which can then be removed by the classical algorithms). □

Now, let $R = R_1 \cup \cdots \cup R_n$ be an extrapolation system where each of the R_i's is an extrapolation rule wrt. a pair $(u_i, k_i) \in \Sigma^* \times \mathbb{N}^{>0}$. Our idea is to define a condition on R such that the computation of $R^*(L)$ can be done for every language L by computing sequentially closures wrt. each of the extrapolation rules R_i in some ordering. Let $\prec \subseteq \Sigma^* \times \Sigma^*$ be the smallest relation such that for every $u, v \in \Sigma^*$, $u \prec v$ if (1) u is not a factor of v (i.e., u does not appear as a subword of v), and (2) u cannot be written as $w_1 v^p w_2$ for any $p \in \mathbb{N}$ and two words w_1, w_2 such that w_1 is a suffix of v and w_2 is a prefix of v. We can prove the following lemma which says that if $u \prec v$, then u can never appear in any power of v.

Lemma 4.3.2 *$\forall u, v \in \Sigma^*$, if $u \prec v$, then $\forall p \geq 0\ \forall w_1, w_2 \in \Sigma^* : v^p \neq w_1 u w_2$.*

Proof. Immediate from the definition of $\prec$: The fact that u can appear in some power of v implies that one of the two conditions defining $u \prec v$ is false. □

We say that the extrapolation system R is *serialisable* if the reflexive closure of the relation $\prec$ (i.e., $\prec \cup id$) defines a partial ordering on the set $\{u_1^{k_1}, \ldots, u_n^{k_n}\}$ (i.e., $\prec$ is antisymmetric and transitive on this set).

Lemma 4.3.3 *Let R be a serialisable extrapolation system and $R_{i_1} R_{i_2} \ldots R_{i_n}$ be a total ordering of the rules of R which is compatible with $\prec$. Then, $R^* = R^*_{i_n} \circ R^*_{i_{n-1}} \cdots \circ R^*_{i_1}$.*

Proof. Follows from Lemma 4.3.2: Closing by some R_{i_j} never creates new rewriting contexts for any of the R_{i_ℓ} with $\ell < j$. □

From the two lemmas 4.3.1 and 4.3.3, we deduce the following fact.

Theorem 4.3.1 *For every serialisable extrapolation system R and for every regular language L, the set $R^*(L)$ is regular and effectively constructible.*

Closure abstractions (even serialisable ones) are not finite-range in general. To see this, consider the infinite family of (finite) languages $L_n = (ab)^n$ for $n \geq 0$ and the extrapolation rule R with $u = a$ and $k = 1$. Then, the images of the languages above form an infinite family of languages defined by $R^*(L_n) = (a^+ b)^n$ for every $n \geq 0$.

Therefore, in the verification framework described in Section 3.5, the use of a closure abstraction α does not guarantee the termination of the computation $\tau_\alpha^*(Init)$. However, as our experiments show (see Section 4.4), the extrapolation principle used in these abstractions is powerful enough to force termination in many practical cases while preserving the necessary accuracy of the analysis of complex properties.

Let us finally mention that the abstraction schema introduced above defines a family of refinable abstractions.

Proposition 4.3.5 *Let us have an extrapolation system R with respect to a set of pairs $\{(u_1, k_1), ..., (u_n, k_n)\}$, let $k'_1, \ldots, k'_n$ be integers such that $\forall i \in \{1, ..., n\} : k'_i \geq k_i$, and let S be the extrapolation system wrt. $\{(u_1, k'_1), \ldots, (u_n, k'_n)\}$. Then, for every language L, we have $S^*(L) \subseteq R^*(L)$. Moreover, if there exists $j \in \{1, ..., n\}$ such that $k'_j > k_j$ and L contains a word with at least k successive occurrences of u_j, then $S^*(L) \subsetneq R^*(L)$.*

4.4 Applications and Experimental Results

We have experimented with a prototype implementation of our techniques on several procedures manipulating linked lists. We have implemented a prototype compiler translating programs into transducers as explained in Section 4.2. As shown in Table 4.1, we have considered procedures for reversing a list, inserting an element into a list at a given position, deleting an element of a list at a given position, merging two lists element-by-element, and the procedure of Bubblesort over a list. Let us note that although these procedures primarily work with simple linear lists, temporarily they may yield several lists sharing their tails or create circular links. Moreover, we have considered working directly with circular lists too, namely a procedure for reversing such lists and a procedure for removing a segment of a circular list (the motivating example of [MYRS05]).

As mentioned in Section 4.2, a store can have several encodings. Thus, to perform the check $\tau_\alpha^*(Init) \cap Bad = \emptyset$ correctly, we require Bad to contain *all* possible encodings of bad stores. In all properties that we consider below, this can easily be achieved.

4.4.1 Checking Consistency of Working with the Dynamic Memory

For all the examples, we have first checked a basic consistency property that consisted in checking that there is no null pointer dereference, no work with undefined pointers, no memory leak (i.e., there does not arise any undeleted and inaccessible garbage), and that the result is a single list pointed to by the appropriate variable. The specification of such a property for a given procedure is easy and can be derived automatically. For the list reversion example, the

set of bad states can be specified using the below extended regular expression[3] where $V = x?\ y?$:

$$(((e+u)\ \Sigma^*) + (\Sigma\ l_8\ \Sigma^*))\ \&\ \neg(n\ l_8 \mid V \mid ((l\ V\ |) + (V \mid l\ V\ (/\ V)^*\ /\ \#\ |)))$$

The expression says that it is bad when we try to do a null pointer dereference or work with an undefined pointer value—this is recognised automatically in the transducers and signalised by the first letter of the resulting configuration set to e. If the first letter becomes u (for unknown), the program cannot be verified using the given number of markers and we have to add some. Finally, it is bad when we reach the final line l_8, and the result is not an empty list (represented by l behind #) nor a single list pointed to by l. We do not care about the values of x and y.

The above property, of course, holds for the correct versions of all the considered procedures. In such a case, our tool provides the user with a safe overapproximation of all the configurations reachable at every line. In this way, we, e.g., automatically obtain the following invariant of the loop of the list reversion procedure:

$$(nl_2 \mid y \mid lx\ |) + (nl_2 \mid y \mid x \mid l(/)^+\#\ |) +$$
$$(nl_2 \mid\ \mid ly \mid x(/)^+\#\ |) + (nl_2 \mid\ \mid\ \mid x(/)^+\# \mid ly(/)^+\#\ |)$$

Roughly, this invariant says that the list is either empty, is pointed to from l, from x, or partially from x and partially from l.

To try out the ability of our techniques to generate counterexamples, we have also tried to examine a faulty version of the list reversion procedure where lines 4 and 5 were swapped. In this case, an error is reported and we are told that from a list with one element (i.e., from a configuration $n\ l_1 \mid x\ y \mid \mid l\ /\ \#\ |$), we can obtain a circular list (a configuration $n\ l_8 \mid y \mid m_t\ l\ x\ /\ m_f\ |$ where m_f and m_t represent the "from" and "to" versions of a marker m). The user can then also trace the program forwards from the initial configuration or backwards from the erroneous one.

4.4.2 Checking More Complex Properties

Further, we have tried to verify some more complex properties of the considered programs. Let us start, e.g., with the Bubblesort procedure. When checking just its basic consistency property, we have completely abstracted away the data values stored in the list and made all the conditional jumps fully nondeterministic. To check that the procedure really sorts, we used a technique inspired by [MS01]. We considered the values of the list elements to be abstracted to being either greater or less than or equal than their successors. The abstracted data values were represented by two special letters (*gt* and *lte*) associated with every list item. We supposed *lte* and *gt* to be distributed

[3] We use "?" to denote zero or one occurrences and "&" to denote intersection.

arbitrarily in the initial configurations. We then checked that the basic consistency property holds and, moreover, the result is a sorted list (i.e., a sequence of elements labelled—up to the last element—by *lte*).[4]

In the case of the merge procedure, we let all elements of the first list be labelled as a elements in the initial configuration and all elements of the other list as b elements. Then we checked that the output list contains a regular mixture of a and b elements.

Finally, for the list reversion and insertion and circular list reversion procedures, we did a fully precise verification of their effect. In the case of list reversion, this means that the output contains exactly the same elements as before, but in a reversed order. For the insertion procedure, the required property is that the output list is precisely the input list up to one new element added into the appropriate place.

To check the above rather strong property, we have proposed a simple, yet efficient technique. Let us explain it on the case of list reversion. In the initial configurations, we let the first and last element be labelled by special labels *bgn* and *end*. Next, we consider as initial all the configurations that can arise from the original initial configurations by attaching two further labels—namely fst and *snd*—to an arbitrary pair of successive elements. The labels are invisible for the unmodified program—they stay attached to their initial elements. Then, to check the desired property, it suffices that every reachable final configuration starts with *end*, ends with *bgn*, and contains a sequence snd/fst. This guarantees that no element can be dropped (then, there would be a way to obtain a configuration without some of the labels), no element can be added (either *end* would not be the first, *bgn* the last, or some snd/fst pair would get separated by another element), and the elements must be rearranged in the given way (otherwise the required resulting ordering of the labels could be broken).

4.4.3 The Results of the Experiments

For each verification example, we applied one instance of the abstractions presented in Section 4.3. For checking the basic consistency properties, we used the piecewise 0-2 counter abstraction with no decomposition symbols ($\Sigma_1 = \emptyset$) and with strong symbols Σ_2 containing the pointer variables, the separator $|$, and the symbol #. Therefore, just the parts of words containing exclusively the / symbols are abstracted. As remarked in Section 4.2.1, / is the only symbol which can appear an unbounded number of times in lists without data. Therefore, our abstraction is finite-range by Proposition 4.3.3. For the more complex properties, we used closure abstractions. The extrapolation rules we applied correspond to the loops one naturally expects to possibly arise in the considered structures (e.g., $(/a, 2)$, $(/b, 2)$, $(/a/b, 2)$ for the list merge

[4] Interestingly, the more precise verification was faster in this case, which is probably due to less non-determinism in the program.

procedure)—providing such information seems to be easy in many practical situations.[5] In all the cases, the abstractions we used are defined by serialisable extrapolation systems. Therefore, by Theorem 4.3.1, they are regular and effectively computable.

We tried out both verification over programs described by a single transducer as well as over programs described by a set of transducers (one per arc of the program control flow graph)—such transducers can easily be obtained by splitting the particular statement transducers according to the line the control should proceed to). Column T of Table 4.1 shows the running times obtained in the latter case. They were about 1.6 to 6 times better than in the former case. The computation times are presented for the minimum number of markers necessary not to run into the “do not known” result. In the case of inserting into a list, we, however, indicate that sometimes it may be advantageous to use more than a necessary number of markers, which is especially the case of loop-free procedures where it may completely eliminate the need for the complex operation of shifting. For every experiment, we also indicate the number of states and transitions of the biggest encountered automaton (or transducer).

We further made a comparison with the abstract regular model checking techniques based on automata abstraction introduced in [BHV04]. We considered the case of programs modelled by a single transducer for which these techniques were implemented. We observed an equal performance on the faulty reverse example, but on the other examples, the new techniques were about 2.9 to 88 times better (not taking into account the Bubblesort example and checking of the correct mixture property for the list merge example where we stopped the tool based on [BHV04] after 2000 seconds).

The verification times obtained from our prototype are very encouraging. Some of the verification times that can be found in the literature for similar verification experiments (especially the ones obtained from PALE) are lower but that is partly due to an incomparable degree of automation (especially in PALE where a significant amount of user intervention is needed) and partly due to the fact that our tool is just an early Prolog-based prototype. We believe that much better times can be expected from a more solid implementation of the tool.

4.5 Summary and Further Work

In the chapter, we have discussed the area of verifying programs manipulating dynamic linked data structures. We have provided an overview of a number of approaches so far proposed in this area and serving as a basis for its further development, which is currently very live. Then, we have concentrated on our

[5] Moreover, it is possible to come up with heuristics that automatically derive the necessary extrapolation rules as mentioned at the end of the section.

Table 4.1. Some results of experimenting with classical and circular linked lists (obtained at 2.4GHz Intel Pentium 4 from an early prototype tool based on Yap Prolog and the FSA library)

Program	Markers	$\lvert M\rvert^{max}_{states+transitions}$	T_{sec}
Reverse, basic consistency	0	51+105	0.3
Reverse, full	0	281+369	4.2
Faulty reverse	1	61+138	0.2
Insert, basic consistency	0	81+102	0.5
Insert, basic consistency	2	165+577	0.15
Insert, full	0	755+936	10.8
Delete, basic consistency	0	55+113	0.3
Merge, basic consistency	0	209+279	2.7
Merge, correct mixture	0	1080+1415	40.4
Bubblesort, basic consistency	2	2095+2872	305
Bubblesort, full	2	2339+2887	279
Circular list reverse, basic consistency	3	655+764	5.4
Circular list reverse, full	3	2349+2822	50.6
Circ. list – removing a segment, bas. cons.	2	116+291	1.0

own approach to automatic verification of programs with dynamic linked data structures built on an exploitation of abstract regular model checking combining automata-based symbolic reachability analysis with abstraction techniques.

Our abstract regular model checking-based approach applies to C-like sequential programs with 1-selector linked structures, for which it allows to verify automatically (safety) properties concerning their data structures. The same techniques can also be used for an automatic invariant generation for these programs. Notice that our approach is not restricted to C programs but can be adapted to other languages with similar operations on linked structures too. The techniques we define are based on simple abstractions of regular sets of configurations which, on one hand, are abstract enough to force termination in many practical cases and, on the other hand, are accurate enough to handle complex properties of the considered data structures. The experimental results are quite encouraging and show the applicability of our approach at least to particular pointer-intensive library routines.

A nice point about the work presented here is that it provides a basis for various possible generalisations. One of them is the use of abstract regular *tree* model checking allowing one to deal with tree-like and even more general dynamic linked data structures (doubly-linked lists, trees with linked leaves, etc.). We have concretised this idea in a work published very recently in [BHRV06b] and mentioned in Chapter 5. The method presented there belongs among the most automated and general approaches to verifying programs manipulating dynamic linked data structures.

In the above, we have also provided two new abstraction schemas usable in the framework of abstract regular model checking—the piecewise 0-k counter abstraction and the closure abstraction. Unlike the sooner proposed representation-oriented abstractions defined on sets of configurations encoded by automata, these new abstractions are defined on particular configurations encoded as words. The techniques are defined in a general way, which makes them not restricted to the application domain we consider here. In fact, they can be used as efficient acceleration techniques in the generic framework of regular model checking for the verification of various classes of infinite-state systems as well. An interesting question for the future is a generalisation of these schemas for the case of abstract regular tree model checking where (as we will see in the next chapter) representation abstractions have so-far been considered only.

A certain deficiency of the closure abstraction technique as presented above is the need to manually provide the extrapolation rules when non-pointer data fields are not abstracted away. However, recently, we have proposed a heuristic for automatically deriving such rules based on on-the-fly monitoring of non-looping sequences of states in the encountered automata and on trying to divide them to a given number of equal subsequences, which can then be used as a basis for extrapolation. This heuristic was successful in all the considered examples with a similar time and space efficiency as presented above (the verification times being sometimes worse but sometimes even better). A proper theoretical as well as practical investigation of this technique is a part of our future work.

In general, in the area of verifying programs manipulating dynamic linked data structures, it is necessary to further improve the existing techniques to be able to handle more general classes of structures (lists and trees with various additional pointers), more complex programs manipulating them (e.g., concurrent and recursive programs with local pointer variables), and harder to verify properties (including termination, quantitative properties like balancedness, and so on). We hope that the techniques presented here and their extension to trees can be useful for this reason when they are further optimised and generalised. The optimisations may include, e.g., some canonisation techniques on automata reducing the number of ways a single store may be encoded, lifting configuration-oriented abstractions to tree regular model checking, trying to avoid the expensive determinisation step by working with non-deterministic automata, etc. The generalisation may consist, e.g., in using more general classes of automata—like, for instance, automata with constraints as in Chapter 6. Another direction is then the development of techniques to handle not only particular pointer-intensive library routines but also larger programs. Here, a crucial issue (apart from further optimisations) is how to combine techniques for dealing with linked data structures with methods being developed for dealing with other data types (as, e.g., predicate abstraction).

5

Regular Tree Model Checking

In this chapter, we discuss issues related to the generalisation of word regular model checking to tree regular model checking in order to handle systems whose configurations have a more complex than a linear (or easily linearisable) structure. The interest in this generalisation is motivated by the fact that tree like structures are very common in computer science and appear naturally in many modelling and verification contexts.

Trees are, for example, very common as the topology underlying various *distributed algorithms* (such as mutual exclusion protocols, leader election protocols, etc.). Labelled trees of an arbitrary height may then be straightforwardly used to represent configurations of the parameterised tree-shaped networks of processes arising in such protocols: a process is a node of a tree and the label represents its control state. Trees also arise naturally as a representation of configurations of *multithreaded recursive programs* (see, e.g., [Esp02, KvS04]) or as a representation of configurations of *cryptographic protocols* (cf., for instance, [GK00, GT01]). In both of the just mentioned cases, the fact that trees naturally correspond to terms is exploited. Further, trees may be as well exploited as a *representation structure of heaps* [KS93] (here, trees may be used directly or as a backbone over which additional pointers may be defined) or when representing structured data such as *XML documents* [BKMW01].

Regular tree model checking (RTMC) exploits the fact that regular tree languages and finite-state tree automata enjoy many similar properties to word regular languages and finite-state automata. In regular tree model checking, configurations of systems have the form of trees over a finite ranked alphabet[1], possibly infinite, but regular sets of configurations are represented using

[1] Sometimes, unranked trees, i.e., trees with a variable arity, are also considered as, e.g., in [Sha01].

(usually bottom-up[2]) finite-state tree automata, and the one-step transition relations are represented using finite-state tree transducers. Subsequently, like in the word regular model checking case, one may try to compute the set of all reachable configurations or the reachability relation. Again, to make the computation terminate as often as possible, various methods are used to accelerate the computation.

Below, we first introduce the basic tree automata notions. Then, we formalise the regular tree model checking framework and we briefly survey the various acceleration mechanisms proposed in this area. Finally, we concentrate on abstract regular tree model checking as a part of our contribution to the research on regular tree model checking [BHRV05, BHRV06a, BHRV06b].

5.1 Tree Automata and Transducers

We now introduce the basic formal notions of regular tree languages and transducers. A more detailed description can be found, e.g., in [CDG+05, Eng75].

Tree Automata and Languages

An *alphabet* Σ is a finite set of symbols. Σ is called *ranked* if there exists a *rank* function $\# : \Sigma \to \mathbb{N}$. For each $k \in \mathbb{N}$, $\Sigma_k \subseteq \Sigma$ is the set of all symbols with rank k. Symbols of Σ_0 are called *constants*. Let χ be a denumerable set of symbols called *variables*. $T_\Sigma[\chi]$ denotes the set of *terms* over Σ and χ. The set $T_\Sigma[\emptyset]$ is denoted by T_Σ, and its elements are called *ground terms*. A term t from $T_\Sigma[\chi]$ is called *linear* if each variable occurs at most once in t. Terms in $T_\Sigma[\chi]$ can be viewed as trees—leaves are labelled by constants and variables, and each node with k sons is labelled by a symbol from Σ_k.

A *bottom-up tree automaton* over a ranked alphabet Σ is a tuple $A = (Q, \Sigma, F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, and δ is a set of transitions of the following types: (i) $f(q_1, \ldots, q_n) \to_\delta q$, (ii) $a \to_\delta q$, and (iii) $q \to_\delta q'$ where $a \in \Sigma_0$, $f \in \Sigma_n$, and $q, q', q_1, \ldots, q_n \in Q$.

Note: Below, we call a bottom-up tree automaton simply a tree automaton.

Let t be a ground term. A run of a tree automaton A on t is defined as follows. First, leaves are labelled with states. If a leaf is a symbol $a \in \Sigma_0$ and there is a rule $a \to_\delta q \in \delta$, the leave is labelled by q. An internal node $f \in \Sigma_k$ is labelled by q if there exists a rule $f(q_1, q_2, \ldots, q_k) \to_\delta q \in \delta$ and the first son of the node has the state label q_1, the second one q_2, ..., and the last one q_k. Rules of the type $q \to_\delta q'$ are called *ε-steps* and allow us to change a state

[2] Nondeterministic top-down tree automata have the same expressive power as bottom-up tree automata, but unlike in the bottom-up case, their deterministic form is strictly less powerful.

label from q to q'. If the top symbol is labelled with a state from the set of final states F, the term t is accepted by the automaton A.

A set of ground terms accepted by a tree automaton A is called a *regular tree language* and is denoted by $L(A)$. Let $A = (Q, \Sigma, F, \delta)$ be a tree automaton and $q \in Q$ a state, then we define the *language of the state* q—$L(A, q)$—as the set of ground terms accepted by the tree automaton $A_q = (Q, \Sigma, \{q\}, \delta)$. The language $L^{\leq n}(A, q)$ is defined to be the set $\{t \in L(A, q) \mid height(t) \leq n\}$.

Tree Transducers

A *bottom-up tree transducer* is a tuple $\tau = (Q, \Sigma, \Sigma', F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is an input ranked alphabet, Σ' is an output ranked alphabet, and δ is a set of transition rules of the following types: (i) $f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow_\delta q(u)$, $u \in T_{\Sigma'}[\{x_1, \ldots, x_n\}]$, (ii) $q(x) \rightarrow_\delta q'(u)$, $u \in T_{\Sigma'}[\{x\}]$, and (iii) $a \rightarrow_\delta q(u)$, $u \in T_{\Sigma'}$ where $a \in \Sigma_0$, $f \in \Sigma_n$, $x, x_1, \ldots, x_n \in \chi$, and $q, q', q_1, \ldots, q_n \in Q$.

Note: In the following, we call a bottom-up tree transducer simply a tree transducer. We always use tree transducers with $\Sigma = \Sigma'$.

A run of a tree transducer τ on a ground term t is similar to a run of a tree automaton on this term. First, rules of type *(iii)* are used. If a leaf is labelled by a symbol a and there is a rule $a \rightarrow_\delta q(u) \in \delta$, the leaf is replaced by the term u and labelled by the state q. If a node is labelled by a symbol f, there is a rule $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow_\delta q(u) \in \delta$, the first subtree of the node has the state label q_1, the second one q_2, ..., and the last one q_n, then the symbol f and all subtrees of the given node are replaced according to the right-hand side of the rule with the variables $x_1, \ldots, x_n$ substituted by the corresponding left-hand-side subtrees. The state label q is assigned to the new tree. Rules of type *(ii)* are called *ε-steps*. They allow us to replace a q-state-labelled tree by the right hand side of the rule and assign the state label q' to this new tree with the variable x in the rule substituted by the original tree. A run of a transducer is successful if the root of a tree is processed and is labelled by a state from F.

A tree transducer is *linear* if all right-hand sides of its rules are linear (no variable occurs more than once). The class of linear bottom-up tree transducers is closed under composition. A tree transducer is called *structure-preserving* (or a *relabelling*) if it does not modify the structure of input trees and just changes the labels of their nodes. By abuse of notation, we identify a transducer τ with the relation $\{(t, t') \in T_\Sigma \times T_\Sigma \mid t \rightarrow_\delta^* q(t')$ for some $q \in F\}$. For a set $L \subseteq T_\Sigma$ and a relation $\varrho \subseteq T_\Sigma \times T_\Sigma$, we denote $\varrho(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in \varrho\}$ and $\varrho^{-1}(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in \varrho\}$. If τ is a linear tree transducer and L is a regular tree language, then the sets $\tau(L)$ and $\tau^{-1}(L)$ are regular and effectively constructible [Eng75, CDG+05].

Let $\iota \subseteq T_\Sigma \times T_\Sigma$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$ and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Below, we suppose $id \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

5.2 RTMC: The Idea and Approaches

As we have already mentioned, regular tree model checking is a generalisation of (word) regular model checking to trees. A configuration of a system is encoded as a term (tree) over a ranked alphabet and a set of such terms as a regular tree automaton. The transition relation of a system is encoded as a linear tree transducer τ. We are given a tree automaton *Init* encoding the set of initial states. For safety properties, a set of bad states (represented by a tree automaton *Bad*) is given.

To illustrate the use of tree automata and transducers, let us consider a simple example—namely, a generalisation of the simple token passing protocol from Sect. 3.2 to trees. We suppose having a tree network of processes of an arbitrary size (for simplicity, with only binary inner nodes). Initially, a token is situated in one of the leaf nodes. Then, it is to be sent up to the root. We would like to check that the token does not disappear nor duplicate.

The initial set of configurations of the simple tree token passing protocol can be modelled by the tree automaton *Init* with the ranked alphabet $\Sigma = \Sigma_0 \cup \Sigma_2$ where $\Sigma_0 = \{T_0, N_0\}$ and $\Sigma_2 = \{T, N\}$ (to respect the formal definition of a ranked alphabet, we distinguish leaf and non-leaf nodes with/without a token), $Q_{Init} = \{p_0, p_1\}$, $F_{Init} = \{p_1\}$, and the following transitions:

$$
\begin{array}{ll}
N_0 \to p_0 & T_0 \to p_1 \\
N(p_0, p_0) \to p_0 & \\
N(p_1, p_0) \to p_1 & N(p_0, p_1) \to p_1
\end{array}
$$

The one-step transition relation may be represented by the tree transducer τ with Σ used as the input/output alphabet, $Q_\tau = \{q_0, q_1, q_2\}$, $F_\tau = \{q_2\}$, and the following transitions[3]:

$$
\begin{array}{ll}
N_0/N_0 \to q_0 & T_0/N_0 \to q_1 \\
N/N(q_0, q_0) \to q_0 & T/N(q_0, q_0) \to q_1 \\
N/T(q_1, q_0) \to q_2 & N/T(q_0, q_1) \to q_2 \\
N/N(q_2, q_0) \to q_2 & N/N(q_0, q_2) \to q_2
\end{array}
$$

Finally, the set of bad configurations may be encoded by the tree automaton *Bad* with Σ as its ranked alphabet, $Q_{Bad} = \{r_0, r_1, r_2\}$, $F_{Bad} = \{r_0, r_2\}$, and the following transitions:

[3] We are dealing with a relabelling transducer and for a better readability, we write its transitions in the form $f/g(q_1, q_2) \to q$ where f is an input symbol and g an output symbol.

$N_0 \to r_0$	$T_0 \to r_1$
$N(r_0, r_0) \to r_0$	$T(r_0, r_0) \to r_1$
$N(r_1, r_0) \to r_1$	$N(r_0, r_1) \to r_1$
$T(r_1, r_0) \to r_2$	$T(r_0, r_1) \to r_2$
N or $T(r_1, r_1) \to r_2$	N or $T(r_0$ or $r_1, r_2) \to r_2$
N or $T(r_2, r_0$ or $r_1) \to r_2$	N or $T(r_2, r_2) \to r_2$

Similarly to the case of classical word regular model checking, the basic *safety verification problem of regular tree model checking* consists in deciding whether

$$\tau^*(L(Init)) \cap L(Bad) = \emptyset. \tag{5.1}$$

Of course, this problem is again in general undecidable, an iterative computation of $\tau^*(L(Init))$ does not necessarily terminate, and so some acceleration techniques are needed to make it terminate as often as often possible. We briefly survey the acceleration mechanisms proposed for regular tree model checking below.

5.2.1 Acceleration in Regular Tree Model Checking

In [Sha01, SP02], a tree generalisation of the approach based on *acceleration schemes* [PS00] is considered. In particular, a generalisation of the *global acceleration of unary transitions* is used. It allows all processes in a tree to fire a certain transition within one accelerated sweep throughout a tree from its leaves to the root (or the other way round). The method has been implemented in the TLV[T] tool and tested on several simple parameterised tree-shaped process networks.

In [BT02], *extrapolation* (widening) over tree automata is discussed. Like in [BJNT00], the approach is based on detecting a growth pattern in a sequence of automata and on accelerating by adding an arbitrary number of occurrences of the growth pattern. An advantage of this method is that it allows for dealing with non-structure preserving tree transducers. On the other hand, a disadvantage is that it was not implemented nor the possibilities of its implementation were considered in detail (the main problem being how to efficiently detect increments, with which one should extrapolate, in tree automata).

The works [AJMd02, ALdA05] propose a generalisation of the *quotienting approach* to tree regular model checking. Structure-preserving tree transducers (also called relabelling transducers) are only considered in this case.

In [ALdA05], quotienting based on equivalences defined via *downward* and *upward simulations* among tree automata states as an analogy of the forward and backward simulations from the word case, which we discussed in Section 3.4.2, is proposed. Let us, however, note that while the notion of downward simulation is quite close to the word case, the upward simulation is different. This is due to the asymmetric nature of trees where when going up from some node, one has to also take into account the context of the considered node.

That is why the upward simulation is defined wrt. downward simulations on all nodes that appear as siblings of the considered node. The general notion of downward and upward simulation is then concretised using *prefix* and *suffix copying* tree automata states again in a somewhat similar way as in the word case (while taking into account the complications related to the upward simulation). The approach was implemented in a prototype way and tested on several case studies from the area of parameterised tree-shaped process networks (the same as those we discuss later in this chapter in relation to the experiments on abstract regular tree model checking).

Below, we describe a generalisation of the abstract regular model checking approach to the tree case we proposed in [BHRV05, BHRV06a]. The approach allows one to deal with structure-preserving as well as non-preserving transducers. Similarly to the word case, the introduction of an *automated abstraction* with a counterexample-guided refinement brings in a quite efficient way for fighting the state explosion problem in the number of tree automata states.

5.3 Abstract Regular Tree Model Checking

Like in abstract word regular model checking, abstract regular tree model checking is based on abstract fixpoint computations in some *finite* domain of automata. The abstract fixpoint computations always terminate and provide overapproximations of the reachability sets. To achieve this, we define techniques that systematically map any tree automaton M to a tree automaton M' from some finite domain such that M' recognises a superset of the language of M. For the case that the computed overapproximation is too coarse and a spurious counterexample is detected, we give effective principles allowing the abstraction to be refined such that the new abstract computation does not encounter the same counterexample.

We, in particular, propose two abstractions for tree automata. Again similarly to abstract word regular model checking, both of them are based on collapsing automata states according to a suitable equivalence relation. The first is based on considering two tree automata states equivalent if their *languages of trees up to a certain fixed height* are equal. The second abstraction is defined by a set of *regular tree predicate languages* as an analogy to the word automata predicate abstraction. However, we also show that not all abstraction schemas from the word case can be lifted to the tree case.

We have implemented the above abstractions in a *prototype tool* using the Timbuk [Gen] tree automata library. We have experimented with the tool on various parameterised tree network protocols. The results are very encouraging and compare very favourably with other tools. This motivated our recent work [BHRV06b] where we use a new implementation of abstract regular tree model checking based on the MONA tree automata libraries [KM01] for verifying programs manipulating complex dynamic linked data structures.

5.3.1 The Framework of Abstract Regular Tree Model Checking

The basic framework of abstract regular tree model checking can be formalised in a way quite similar to word regular model checking (unlike other approaches to regular tree model checking where a rather complex, new theoretical framework was needed to obtain them as a generalisation of the appropriate word cases). We basically phrase all the needed concepts not for classical finite automata, but for for finite tree automata.

Let Σ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over Σ. We define an abstraction function as a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Given a tree transducer τ and abstraction α, we define a mapping $\tau_\alpha : \mathbb{M}_\Sigma \to \mathbb{M}_\Sigma$ as $\forall M \in \mathbb{M}_\Sigma : \tau_\alpha(M) = \hat{\tau}(\alpha(M))$ where $\hat{\tau}(M)$ is a minimal automaton describing the language $\tau(L(M))$. An abstraction α is *finite range* if the set $\mathbb{A}_\Sigma$ is finite.

Let $Init$ be a tree automaton representing the set of initial configurations and Bad be a tree automaton representing the set of bad configurations. Now, we may iteratively compute the sequence $(\tau_\alpha^i(Init))_{i\geq 0}$. Since we suppose $id \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(Init) = \tau_\alpha^k(Init)$. The definition of α implies $L(\tau_\alpha^k(Init)) \supseteq \tau^*(L(Init))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(Init))$.

If $L(\tau_\alpha^k(Init)) \cap L(Bad) = \emptyset$, then the verification problem denoted as 5.1 above has a positive answer. Otherwise, the answer to the problem 5.1 is not necessarily negative since during the computation of $\tau_\alpha^*(L(Init))$, the abstraction α may introduce extra behaviours leading to $L(Bad)$. Let us examine this case. Assume that $\tau_\alpha^*(Init) \cap L(Bad) \neq \emptyset$, which means that there is a symbolic path:

$$Init,\ \tau_\alpha(Init),\ \tau_\alpha^2(Init), \cdots \tau_\alpha^{n-1}(Init),\ \tau_\alpha^n(Init) \tag{5.2}$$

such that $L(\tau_\alpha^n(Init)) \cap L(Bad) \neq \emptyset$. We analyse this path by computing the sets $X_n = L(\tau_\alpha^n(Init)) \cap L(Bad)$, and for every $k \geq 0$, $X_k = L(\tau_\alpha^k(Init)) \cap \tau^{-1}(X_{k+1})$. Two cases may occur: (i) either $X_0 = L(Init) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, which means that the problem 5.1 has a *negative answer*, or (ii) there is a $k \geq 0$ such that $X_k = \emptyset$, and this means that the symbolic path 5.2 is actually a *spurious counterexample* due to the fact that α is too coarse. In this last situation, we need to refine α and iterate the procedure. Therefore, our approach is based on the definition of abstraction schemas allowing to compute families of (automatically) refinable abstractions.

5.3.2 Abstractions over Tree Automata

Below, we discuss two possible tree automata abstraction schemas which are based on tree automata state equivalence. First, tree automata states are

split into several equivalence classes by an equivalence relation. Then, the abstraction function collapses states from each equivalence class into one state. Formally, a tree automata state equivalence schema $\mathbb{E}$ is defined as follows: To each tree automaton $M = (Q, \Sigma, F, \delta) \in \mathbb{M}_\Sigma$, an equivalence relation $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ is assigned. Then the automata abstraction function $\alpha_{\mathbb{E}}$ corresponding to the abstraction schema $\mathbb{E}$ is defined as $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ finitary if $\alpha_{\mathbb{E}}$ is finitary (i.e., there is a finite number of equivalence classes). We refine $\mathbb{E}$ by making $\sim_M^{\mathbb{E}}$ finer.

Abstraction Based on Tree Languages of Finite Height

We now present the possibility of defining automata state equivalence schemas based on comparing automata states wrt. a certain bounded part of their languages. The abstraction schema $\mathbb{H}_n$ is a generalisation of the schema based on languages of words up to a certain length proposed for word automata in [BHV04] and described in Sect. 3.5.4. The $\mathbb{H}_n$ schema defines two states of a tree automaton M as equivalent if their languages up to the given height n are identical.

Formally, for a tree automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{H}_n$ defines the state equivalence as the equivalence $\sim_M^n$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

There is a finite number of languages of trees with a maximal height n, and so this abstraction is finite range. Refining of the abstraction can be done by increasing the value of n.

The abstraction schema $\mathbb{H}_n$ can be implemented in a similar way as minimisation of tree automata [CDG+05]. Just the main loop of the minimisation procedure is stopped after n iterations.

Abstraction Based on Predicate Tree Languages

We next introduce a predicate-based abstraction schema $\mathbb{P}_{\mathcal{P}}$, which is inspired by the predicate-based abstraction on words from [BHV04] discussed in Sect. 3.5.3.

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma, F, \delta)$ be a tree automaton, then two states $q_1, q_2 \in Q$ are equivalent if their languages $L(M, q_1)$ and $L(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$.

Formally, for an automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{P}_{\mathcal{P}}$ defines the state equivalence as the equivalence $\sim_M^{\mathcal{P}}$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^{\mathcal{P}} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Clearly, since $\mathcal{P}$ is finite and there is only a finite number of subsets of $\mathcal{P}$ representing the predicates with which a given state has a nonempty intersection, $\mathbb{P}_{\mathcal{P}}$ is *finitary*. This schema can be refined by adding new predicates

into the set $\mathcal{P}$. The following theorem shows that we may eliminate a spurious counterexample by extending the predicate set $\mathcal{P}$ by the languages of all states of the tree automaton representing X_{k+1} in the analysis of the spurious counterexample (recall that $X_k = \emptyset$) as presented in Section 5.3.1.

Theorem 5.3.1 *Let us have any two tree automata $M = (Q_M, \Sigma, F_M, \delta_M)$ and $X = (Q_X, \Sigma, F_X, \delta_X)$ and a finite set of predicate automata $\mathcal{P}$ such that $\forall q_X \in Q_X : \exists P \in \mathcal{P} : L(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) = \emptyset$ too.*

Proof. The proof is a generalisation of the proof of Theorem 3.5.1 stated for word automata in [BHV04] (cf. Section 3.5.3). We prove the theorem by contradiction. Suppose $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) \neq \emptyset$. Let $t \in L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$. As t is accepted by $\alpha_{\mathbb{P}_\mathcal{P}}(M)$, M must accept it when we allow it to perform a certain number of "jumps" between states equal wrt. $\sim_M^\mathcal{P}$—after accepting a subtree of t and getting to some $q \in Q_M$, M is allowed to jump to any $q' \in Q_M$ such that $q \sim_M^\mathcal{P} q'$ and go on accepting from there (with or without further jumps).

Let $i > 0$ be the minimum number of jumps needed for accepting a tree from the set $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ in M, and let t' be such a tree. When looking at the acceptance of t' in M (with some jumps allowed), we can identify maximum subtrees of t' that may be accepted without jumps—in the worst case, they are just the leaves. Let us take any of such subtrees. Such a subtree t_1 is accepted in some q_1, from which M jumps to some q_2 and goes on accepting the rest of the input. Suppose that t_1 is accepted in some $q_X \in Q_X$ in X. As $t_1 \in L(M, q_1)$, $L(M, q_1) \cap L(P) \neq \emptyset$ for the predicate $P \in \mathcal{P}$ for which $L(P) = L(X, q_X)$. Moreover, as $q_1 \sim_M^\mathcal{P} q_2$, $L(M, q_2) \cap L(P) \neq \emptyset$ too. This implies there exists $t_2 \in L(P)$ such that $t_2 \in L(M, q_2)$ and $t_2 \in L(X, q_X)$. However, this means that the tree t'' that we obtain from t' by replacing its subtree t_1 with t_2 and that clearly belongs to $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ can be accepted in M with $i - 1$ jumps, which is a contradiction to the assumption of i being the minimum number of jumps needed. □

Similarly to the word case, the abstraction $\alpha_{\mathbb{P}_\mathcal{P}}$ of a tree automaton M wrt. the state equivalence based on tree predicate languages $\mathcal{P}$ can be implemented as labelling each state of M by the predicates with which its language has a non-empty intersection, and then collapsing states with an equal labelling. When refining $\mathbb{P}_\mathcal{P}$, it is not necessary to store each of the newly introduced predicates corresponding to the states of X_{k+1} independently and then perform the labelling independently for each of them. We may keep just X_{k+1} and then perform labelling not by just X_{k+1} but by each of its states. Moreover, this labelling may be implemented by one simultaneous run through M and X_{k+1}, which corresponds to an efficient simultaneous labelling by all the predicates contained in X_{k+1}.

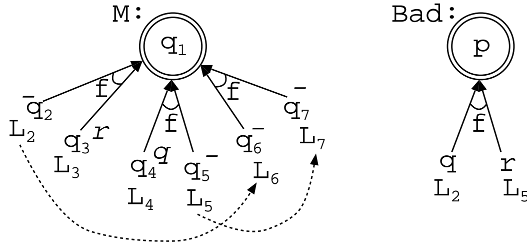


Fig. 5.1. A problem with the forward tree predicate abstraction

Let us add that $\mathbb{P}_{\mathcal{P}}$ has been in particular inspired by the backward predicate-based abstraction schema $\mathbb{B}_{\mathcal{P}}$ from [BHV04], which considers words between a given automaton state and the initial state. It is interesting that as we illustrate in Figure 5.1, it is not possible to obtain a tree analogy with the forward predicate-based abstraction schema $\mathbb{F}_{\mathcal{P}}$ of the word abstract regular model checking, which considers words between a given automaton state and the final states. The tree analogy would be to label a state with a predicate state if the languages of their contexts—i.e., trees where we substitute Σ^* for the language of the node being labelled/used for labelling—have a non-empty intersection. However, in this way, we may not ensure that the abstraction of M from Figure 5.1 will be disjoint with *Bad* (in Figure 5.1, the upper index of the states of M shows by which states of *Bad* they are labelled).

5.3.3 Experiments with Abstract Regular Tree Model Checking

In order to be able to practically evaluate the proposed methods of abstract regular tree model checking, we have implemented them in a prototype tool. We have based our prototype tool on the *Timbuk* library [Gen] written in OCaml. Timbuk provided us with the basic operations over tree automata needed in abstract regular tree model checking (such as union, intersection, complementation, etc.). However, we had to extend Timbuk with a support for tree transducers (and also minimisation, which was not provided by the library). We added two implementations of tree transducers—a simpler and more efficient for structure-preserving transducers and a more complex for general transducers. The latter implementation exploits a decomposition of a tree transducer into three less complicated ones as described in [Eng75]. This decomposition can be performed automatically for any tree transducer.

We have tested our verification methods on several examples of protocols using a parameterised tree-shaped network cited in the literature [KMM+01, ABH+97, AJMd02, ALdA05] where the necessity to cover all possible values of the parameters leads to dealing with infinite state spaces:

- *Simple Token Protocol.* Our running example from Sect. 5.2: A token is being passed in a tree-shaped network from a leaf to the root. We check that the token does not disappear nor replicate.
- *Two-Way Token Protocol.* An analogy to the previous example, but we allow the token to be passed upwards as well as downwards.
- *Percolate Protocol.* A tree-shaped network of processors computes the logical disjunction of the boolean values that appear in the leaf nodes. We check that the computed value is always correct.
- *Tree Arbiter Protocol.* A tree-shaped network is used to implement mutual exclusion among the leaf processors. A request to enter the critical section is propagated upwards till a node is found which has a token allowing one to enter the critical section or which knows where the token is (because it granted the token to one of its children). A node with the token can always send the token upwards or grant it to any of its children. We check the mutual exclusion property.
- *Leader Election Protocol.* One of a set of processors is to be elected a leader and a tree-shaped network is used for this purpose. The leaves are divided into candidates and non-candidates. The information about the existence of candidates is propagated upwards. In the subsequent downward phase, a path leading from the root to one of the candidate nodes is nondeterministically selected and thus a leader is established. We check that exactly one leader is chosen.

All the above examples work with a tree-shaped network of a fixed structure. In order to test the ability of our method to work with non-structure-preserving systems, we have considered a *simple broadcast protocol.* In the protocol, the root sends a message to all leaf nodes. They answer and the answers are combined when travelling upwards. An intermediate node may decide to resend the message downwards and wait for new data. New nodes may dynamically join the network at leaves and also leave the network in a suitable moment. We check that there is at most one active message on each path from the root to the leaves.

The results of our experiments are summarised in Table 5.1. We performed experiments with both the finite-height abstraction as well as with the predicate-based abstraction. We considered both forward as well as backward verification—i.e., starting with the set of initial states and checking that the bad states cannot be reached or vice versa. In the table, we always present the better result of these two approaches. For the finite-height abstraction, we considered the initial height one (and increased it by one if necessary—in the cases presented in Table 5.1, this was not necessary). For the predicate-based abstraction, we considered the automaton describing the set of bad states as the only initial predicate (or—more precisely—all the automata that can be obtained from it by considering each of its states as the only accepting one; in the cases presented in Table 5.1, no refinement was necessary when using these initial predicates). We experimented with the empty initial set of pred-

icates too—this turned out to be the fastest option for the Percolate protocol (one refinement was necessary in this case).

Table 5.1. Some results of experimenting with abstract regular tree model checking

Protocol	$\mathbb{H}_n$	$\mathbb{P}_{\mathcal{P}}$
Token passing	backwards: 0.08s	forwards: 0.06s
Two-way token passing	backwards: 1.0s	forwards: 0.09s
Percolate	backwards: 20.8s	forwards: 2.4s
Tree arbiter	backwards: 0.31s	backwards: 0.34s
Leader election	backwards: 2.0s	forwards: 1.74s
Broadcasting	backwards: 9.1s	forwards: 1.0s

The verification times presented in Table 5.1 were obtained on an Intel Centrino 1.6GHz machine with 768MB of memory. These results are very competitive compared to the other existing approaches to regular tree model checking. Moreover, as mentioned below, we recently implemented a new version of the abstract regular tree model checking framework under the Mona tree automata library [KM01], which provides even much better results though still offering a significant space for further improvements.

5.4 Summary and Further Work on RTMC

We have presented regular tree model checking as an extension of the basic framework of regular model checking. Regular tree model checking has many possible applications as trees (terms) are very common in various domains of computer science and engineering. We have briefly discussed various existing approaches to regular tree model checking and we have in detail presented our approach of *abstract* regular tree model checking.

The first experimental results obtained from our Timbuk/OCaml-based prototype implementation of abstract regular tree model checking are very encouraging. Very recently [BHRV06b], they lead us to an implementation of the method in a new prototype based on the Mona tree automata library [KM01]. This implementation is, in particular, intended for a use within verification of programs manipulating *complex dynamic linked data structures* (doubly-linked lists, trees, trees with additional pointers, etc.). Configurations of these programs, which have the form of general graphs, are encoded over a *tree backbone* using the so-called *routing expressions* to express links that cannot be coded directly in the tree backbone. The encoding is partly similar to the one of PALE [MS01] (cf. Section 4.1.1), but unlike in PALE, the method is fully automated—no loop invariants are to be provided by the user (moreover, we do not require the routing expressions to have a deterministic target, and

the meaning of next pointers to be fixed in advance). Sets of the tree backbones as well as the routing expressions and their manipulation are encoded using tree automata and tree transducers. This approach is a generalisation of the approach of using classical word abstract regular model checking for verification of programs with 1-selector dynamic data structures [BHMV05] discussed in Chapter 4. The results obtained in [BHRV06b] compare quite favourably with other existing approaches for verification of programs with complex dynamic linked data structures (which we briefly described in Chapter 4). In fact, we obtain one of the most general and at the same time most automated approaches for verifying the considered kind of programs having at the same time a reasonable performance (at least for verifying particular pointer-intensive library routines or modules).

Let us note that the above results were achieved despite there is still a lot of space for further improvements of the tree regular model checking techniques and their implementation used in [BHRV06b]. In particular, one can think of *more specialised abstractions* for the domain of programs with dynamic linked data structures than the general-purpose ones that we presented above and that were used in [BHRV06b]. (As mentioned in Chapter 4, using specialised abstractions for word abstract regular model checking within verification of programs with 1-selector dynamic data structures in [BHMV05] lead to speedups up to even two orders of magnitude.) Further, we should, for instance, try to exploit the concept of *guided tree automata* [BKR97] that is usually claimed as one of the main keys to an efficient use of Mona but which has not been used in the early prototype built in [BHRV06b]. These issues belong among the subjects of our further work in this domain. Moreover, we also think of combining the framework with some *non-regular features*, e.g., by introducing various constraints over the tree automata to be able to track more precisely the size of various parts of the memory configurations encoded in an abstract way in trees. This should allow us to verify also *quantitative properties* (like, e.g., balancedness) and/or *termination properties* of the considered programs (a certain way illustrating how such measures could be associated with trees will be discussed in Chapter 6 although in a less general and only semi-automated framework).

Further interesting directions for future work include research on other promising application areas of (abstract) regular tree model checking. Among them, we can mention, for instance, verification of XML manipulations. Indeed, XML documents have a tree structure, and various results and tools for XML handling are based on tree automata or hedge automata (i.e., unbounded-width tree automata where one works with regular sets of predecessor states of a given automaton state) [BKMW01]. Furthermore, we can consider an application of the described framework for programs with cryptographic protocols along the lines of [Mon03]. For all these applications, one should study their suitable encoding in tree automata and transducers and the possibility of defining application dependent abstractions.

6

Tree Automata with Size Constraints

Apart from generalising word regular model checking to tree or omega regular model checking, one can also think of extending the framework to dealing with various classes of (word/tree/omega) *non-regular languages*, for which one can find a suitable finite encoding. The chosen classes of languages (and the associated encoding) should allow one to—if possible, efficiently—implement some form of transduction, to use the needed language operations and tests (depending on the exact setting, one may need union, intersection, complement, emptiness checking, and/or inclusion checking), and to accelerate the computation. Finding a super-class of regular languages that meets such criteria is, however, not easy even though sometimes, some of the requirements may be relaxed: For example, instead of having general transducers, special purpose algorithms for implementing various types of transitions of the considered systems over the chosen (automata) representation may be provided. Some of the language operations and tests may sometimes be avoided—e.g., we may avoid inclusion testing when we strengthen the fixpoint tests (for termination of the reachability analysis) via checking identity on the chosen representation of languages. Acceleration may be provided in a form specialised to a certain class of systems, or it can be avoided when we consider only loop free systems or when we split loops by manually provided loop invariants.

Despite the above mentioned difficulties, a number of symbolic verification approaches based on dealing with non-regular languages have been proposed. Below, we first briefly discuss some of the most interesting among them. Then, we illustrate in detail what it means to go beyond dealing with regular state spaces on our work [HIV06] in which we introduce a class of *tree automata with size constraints* and their application to verification of *programs manipulating dynamic balanced tree structures*.

6.1 Works Trying To Go Beyond RMC

CQDDs

Queue-content decision diagrams (QDDs) have been proposed in [BP96] as a symbolic representation of the queue contents of systems communicating via non-lossy unbounded FIFO queues (channels). QDDs are common finite-state automata accepting words that are a concatenation of the contents of the queues used in a given system (their alphabets are considered disjoint). In [BP96], a special-purpose algorithm over QDDs is proposed for computing the exact effect of repeating any number of times a certain kind of loops (preserving regularity) in the systems communicating via queues. This algorithm can be used to accelerate the computation of reachable states in this class of systems as follows: The user (or some heuristic) chooses some loops of the given system that are used as a basis of the so-called "*meta-transitions*" which are added to the system and whose task is to atomically produce the effect of firing the appropriate loops any number of times. Then, all the transitions are repeatedly fired (using the proposed algorithm to compute the effect of firing meta-transitions) till a fixpoint is reached. Alternatively, general-purpose regular model checking can of course be used too.

In [BH99], *constrained queue-content decision diagrams* (CQDDs) have been proposed to capture non-regularities in sets of reachable queue contents. A CQDD is based on *restricted finite-state automata extended with linear (Presburger) constraints*[1] *on the number of occurrences of transitions of the automata appearing in accepting runs*. The restriction among others excludes nested loops in the automata underlying CQDDs—these automata are deterministic and accept words of the form $u_1 v_1^* u_2 v_2^* ... u_m v_m^* u_{m+1}$ where all the u_is and v_is are words over a given alphabet Σ such that only u_1 and u_{m+1} may be empty. CQDDs enjoy very nice automata-theoretic properties (closure wrt. union, intersection, concatenation, decidability of emptiness, membership, inclusion, etc.). CQDDs allow one to exactly characterise the effect of iterating any loop in a system communicating via queues. For this purpose, a special-purpose algorithm is proposed in [BH99] that can be used in a meta-transition-based reachability analysis of systems communicating via queues.

ℤ-input 1-Counter Machines

Model checking linear-time temporal logic over push-down systems[2] is known to be polynomial for a fixed formula [BEM97, FWW97]. For encoding the set of configurations reachable by a push-down system, which is guaranteed

[1] Presburger arithmetics [Pre29] is a first-order formal arithmetics with addition and comparison on variables ranging over integers. Deciding of formulae of the Presburger arithmetics is 2EXPSPACE-complete in the size of the formulae.

[2] Note that push-down systems unlike push-down automata do not have an input.

to be regular, one can use a symbolic representation based on a finite-state automaton with multiple initial states. Then, the current control state of the push-down system corresponds to an initial state of the finite-state automaton, and the current contents of the stack is a word that must be accepted from this state. The automaton encoding all reachable configurations may be obtained by the so-called *saturation*, i.e., by adding new transitions (and sometimes also states) to the finite-state automaton encoding the initial set of configurations. For instance, when computing Pre^* of some set, if there is a rule $\langle p, \gamma \rangle \rightarrow \langle p', w \rangle$ in the push-down system being model checked (for $\gamma \in \Gamma$, $w \in \Gamma^*$ where Γ is the stack alphabet) and if $p' \stackrel{w}{\rightarrow}^* q$ in the finite-state automaton A encoding the so-far computed reachability set of the push-down system, we add a transition $p \stackrel{\gamma}{\rightarrow} q$ to A.

In [BHM03], the above approach is generalised to dealing with recursive procedures having one integer parameter. For encoding sets of reachable configurations of such systems, the so-called *$\mathbb{Z}$-input 1-Counter Machines* are used. The input of such machines is a sequence $X_1(k_1)X_2(k_2)...X_n(k_n)$ where X_i are symbols from some finite alphabet and $k_i \in \mathbb{Z}$ for $i \in \{1, ..., n\}$. The machines can increment and decrement their counter and branch according to Presburger tests on the counter or according the symbol X_i read and a comparison of the associated integer parameter k_i with a constant or the value of the counter. On top of such machines, a reachability procedure based on saturation (together with some modifications of the involved Presburger formulae) is proposed. This way all configurations reachable from a set encoded by a $\mathbb{Z}$-input 1-counter machine or all configurations backward reachable from a regular set can be obtained. Moreover, a method is proposed for model checking a special logic allowing one to test reachability of certain specific configurations. These configurations are specified in a form divided into a fixed number of regular patterns with a Presburger constraint on the integer parameters that appear between these patterns. For this purpose, another interesting class of automata is used—namely, the so-called *push-down counter automata with a finite number of reversal bounded counters* (i.e., counters whose manipulation can switch between incrementing and decrementing a bounded number of times only) [Iba78].

Reversal Bounded Counter Automata

The above mentioned automata with a finite number of *reversal bounded counters* belong—due to having decidable important properties like language emptiness or containment—among the most general, currently known classes of automata suitable for symbolic automata-based verification. In works including [Iba78, ISD$^+$02], various kinds of such automata are studied. They differ in being one-way or two-way (i.e., allowing the head on the input tape move only ahead or also change the direction), in being deterministic or nondeterministic, in having an additional unrestricted counter or a push-down stack, in having different kinds of tests on the counters attached to their tran-

sitions (up to linear-relation tests with parameterised constants), etc. The decidability results on these automata are often obtained via using decidability of Presburger arithmetics.

Reversal bounded counter automata may be directly used for *modelling systems* to be analysed which allows one to exploit the decidability results on safety and reachability on these automata (that build on representing the reachability set of the automata using Presburger arithmetics). Apart from this, reversal bounded counter automata may be used to *represent reachability sets* of various other models.

In the previous subsection, a push-down reversal bounded counter automaton (obtained via a special-purpose construction applied on a $\mathbb{Z}$-input 1-counter machine encoding all reachable configurations of a system of recursive procedures with one integer parameter) is used to describe the subset of the reachable configurations satisfying a certain formula. In [Iba00, IDP03], special-purpose constructions are used to derive reachability relations of systems consisting of *two synchronous (or loosely synchronous—i.e., synchronous up to a bounded difference) discrete timed automata connected via a communication queue* (with some possible extensions). The relations are obtained in the form of two-tape push-down reversal bounded counter automata. Further, in [DBIK04], another special purpose construction is proposed for deriving reachability sets of a class of discrete timed systems modelled by the so-called *past push-down timed automata* whose transition enabling conditions can refer in a certain way to past values of the control-part of these automata.

Automata-based Verification of Networks of Recursive Processes

Specialised counter tree automata as well as various kinds of tree automata with constraints have also been used in the context of analysing concurrent networks of processes with dynamic instantiation and recursion.

For example, in [BT03], process networks are modelled using *process rewrite systems* (PRS) [May00a]. PRS allows for dynamic creation of processes and their concurrent execution via having features of multiset rewriting, and it also allows for sequential execution with recursion via having features of prefix rewriting.[3] In [BT03], various results on analysing PRS are given,

[3] The syntax of a PRS process term t is defined as $t ::= 0 \mid X \mid t.t \mid t \| t$ where 0 is the idle process, X is a process constant, and . and $\|$ are the sequential and parallel composition operators, respectively. The idle process 0 is a neutral element for both . and $\|$, . is associative, and $\|$ is associative and commutative. A PRS is given by a set of rewrite rules $t \to t'$ for terms t, t'. If there is a rewrite rule $t \to t'$, one can rewrite t to t', $t.t''$ to $t'.t''$, and $t \| t''$ to $t' \| t''$ where t, t', t'' are PRS terms. A number of various different approaches to analysing PRS and their different subclasses have been published in the literature—see, e.g., works of R. Mayr, A. Bouajjani, J. Esparza, T. Touili, A. Kučera, P. Jančar, J. Srba, P. Schnoebelen, and others. The mentioned neutrality, associativity, and commutativity involved in PRS induce certain equivalences on sets of PRS terms, which complicates symbolic, automata-based analyses of PRS and may lead to a need of dealing with non-regular languages.

most of them based on using regular tree languages (encoding sometimes not reachability sets but sets of representatives of equivalence classes of the reachability sets). However, a special purpose construction based on a class of the so-called *0-test counter tree automata* is also proposed for computing the set of representatives of equivalence classes of the backwards reachability set for a subclass of PRS called PAD (with no parallel composition on the left-hand side of rules). The 0-test counter tree automata can increment their counters by an integer constant and test them to be all zero. They are effectively closed under intersection with a regular tree language, and their emptiness is decidable.

Next, in [BT05], a generic procedure for computing reachability sets of PRS is provided. It is parameterised by procedures for analysing prefix and multiset rewrite systems and represents reachability sets using a class of *commutative hedge automata*. Commutative hedge automata accept sets of trees of an unbounded width. They generalise classical bottom-up tree automata by having rules $f(L) \rightarrow q$ where L is a regular language on states of the automaton that can appear on the left-hand side of the rule (these rules allow one to cope with associativity), and rules of the form $f(\varphi) \rightarrow q$ where φ is a Presburger formula on the number of occurrences of particular states of the automaton in the left-hand side of the rule (these rules allow one to cope with associativity and commutativity).[4] Commutative hedge automata are effectively closed under Boolean operations, and the emptiness problem is decidable for them. In [BT05], a saturation procedure over these automata (using subprocedures for analysing prefix and multiset rewriting) is proposed.

Furthermore, in [BMOT05], the so-called *constrained dynamic push-down networks* (CDPNs) are proposed as a new model for analysing multithreaded recursive programs. A CDPN can be seen as a collection of identical sequential processes that run in parallel, can create new processes (which become their children), perform pushdown operations, and observe in a certain way the behaviour of their children—a father process can check whether the states of his children (ordered according to their age) belong to a certain (restricted) regular language. The work uses a saturation procedure over *hedge automata* (unbounded-width tree automata) to compute the backwards reachability set starting with a regular set of target configurations. However, the work also shows that the forwards reachability set is non-regular and proposes a method for characterising it using a *context-free grammar*. A similar approach is then considered in [BESS05] for bounded reachability analysis of *asynchronous dynamic pushdown networks* communicating via shared memory.

[4] Similar automata appear (in a combination with a use of a monadic-second order logic) in the reachability analysis for a certain higher-order extension of PRS (allowing one to cope with nested stacks, i.e., stacks of stacks, etc.) considered in [Col03]. Even more powerful automata appear, e.g., in [SSM03, Lug05] in the context of XML querying.

AC-Tree Automata

Associativity and commutativity, which complicate automata-based analyses of networks of processes represented by PRS, show up in other areas too. One of the quite well-known among them is the area of verifying *cryptographic protocols*. Configurations of such protocols may often be viewed as terms and sets of such terms as tree languages. To cope with associativity and commutativity, one can use regular tree approximations as, e.g., in [GK00] linked with the Timbuk tool [Gen]. Another approach has been proposed in a series of works related to the ACTAS tool [OT05]. ACTAS is based on dealing with the so-called *AC-tree automata* being a special case of *equational tree automata* which are a combination of equation systems over the terms being accepted and tree automata (allowing the kind of transitions described in Section 5.1 and also transitions of the form $f(q_1, ..., q_n) \to f(q'_1, ..., q'_n)$) [Ohs01]. AC-tree automata enjoy nice automata-theoretic properties: They are effectively closed under union and intersection, and the membership and emptiness problems are decidable for them. Moreover, a subclass of the so-called *regular AC-tree automata* (without the special type of rules mentioned above) is effectively closed under negation, and containment is decidable over it too. ACTAS uses AC-tree automata to represent sets of configurations reachable in a system (e.g., a cryptographic protocol) described by a term rewriting system. To make the computation of the reachable configurations finish, ACTAS uses several under- and over-approximation techniques (limiting the number of rewrite steps, limiting the depth and width of searching when checking intersection of languages, which is needed when dealing with rewrite rules of the form $f(x, x) \to x$ to check overlapping on the left-hand side, etc.).

Restricted Deterministic Push-Down Automata

In [FP01], an approach that is perhaps the closest in spirit to general-purpose regular model checking is proposed. It is based on a novel subclass of the class of languages of deterministic push-down automata. The class of full deterministic push-down languages itself is problematic for a use in symbolic model checking as it is not closed wrt. projection needed for transductions and there is no known efficient algorithm for checking language equivalence for it.

The new class of languages proposed in [FP01] is based on *push-down automata decomposed* into a *1-state deterministic push-down stack manipulator* and a *finite-state control* that *synchronise* by reading the same input symbols and, moreover, the finite-state part reads the top stack symbols being manipulated too. For this class, a semi-algorithm for projection is proposed in [FP01], the equivalence problem for this class turns out to be efficiently decidable, and the class also retains the original positive properties of deterministic push-down languages. Moreover, if the synchronisation between the push-down manipulator and the finite-state control is solely via the input

symbols, and the stack manipulator used for specifying the initial set of configurations is preserved by the system being verified (which can be checked using the sooner mentioned semi-algorithm for projection), transductions may be done only on the finite-state control part. Further, even acceleration may then be done only on the finite-state part allowing one to use the classical acceleration methods for regular model checking.

In [FP01], the use of the method is illustrated mainly on verifying an abstract version of the Peterson mutual exclusion protocol that cannot be handled via regular model checking.

6.2 Tree Automata with Size Constraints

We now illustrate in more detail an application of more than regular languages (and the associated automata) for symbolic verification on the use of the so-called *tree automata with size constraints* (TASC). In particular, TASC have been proposed in [HIV05, HIV06] for verification of programs manipulating balanced tree structures.

Balanced tree-like search data structures are very often applied to implement in an efficient way lookup tables, associative arrays, sets, or similar higher-level structures, especially when they are used in critical applications like real-time systems, kernels of operating systems, etc. Therefore, there arose a number of such search tree structures like AVL trees, red-black trees, splay trees, and so on [CLR90].

When one intends to use tree automata for symbolically representing sets of configurations reachable by programs manipulating balanced tree structures, one faces two problems. First, as we have already discussed in Chapter 4, the programs can temporarily break the tree shape of the structures being manipulated. This can be solved, e.g., by using routing expressions over a tree backbone to express the links that are not tree-like as in [MS01, BHRV06b] (and as also mentioned in Chapters 4 and 5). In the following, however, we adopt a simpler approach by observing that many algorithms handling balanced tree structures [CLR90] use *tree rotations* (plus the low-level addition/removal of a node to/from a tree) as the only operations that effectively change the structure of the input tree. We consider here such operations as atomic—we suppose their implementation to be checked independently using some of the techniques for dealing with general pointer structures mentioned in Chapter 4. Nevertheless, a generalisation of our framework to handle all the "pointer surgery" in a uniform way is an interesting subject for further research.

A second problem with the use of tree automata in the given area is the fact that classical tree automata as defined in Chapter 5 represent *regular* sets of trees. However, when one needs to reason in terms of *balanced* trees, as in the case of AVL and red-black tree algorithms, one has to reason about

non-regular sets of trees. It is exactly this problem for coping with which we introduce our tree automata with size constraints.

TASC are tree automata whose actions are triggered by arithmetic constraints involving the *sizes* of the subtrees at the current node. The size of a tree is a numerical function defined inductively on the tree structure as, for instance, the height, the maximum number of black nodes on all paths, etc. The main advantage of using TASC in symbolic program verification is that they recognise non-regular sets of tree languages such as *AVL trees*, *red-black trees*, and in general, specifications involving arithmetic reasoning about the lengths (depths) of various (possibly all) paths in the tree. We show that the class of TASC is effectively closed under the operations of union, intersection, and complement. Also, the emptiness problem is decidable for TASC. Moreover, the semantics of programs performing tree updates (node recolouring, rotations, pointer navigation, and appending/removal of nodes) can be effectively represented as changes on the structure of TASC.

In our *verification approach* based on TASC, the user has to provide the precondition and postcondition of the (sequential) imperative program being verified as well as loop invariants for all loops present in the program. As the loop invariants are provided, we can cut the program into loop-free fragments that can be handled separately. The verification problem then consists in checking validity of Hoare triples of the form $\{P\}C\{Q\}$ where P and Q are TASC-encoded sets of configurations corresponding to the precondition or postcondition of the program or to some loop invariant, and C is a loop-free fragment of the program to be verified. Next, we reduce this verification problem to the TASC language emptiness problem.

We validate our approach on an example of the insertion algorithm for red-black trees for which we verify that for a balanced red-black tree input, the output of the insertion algorithm is also a balanced red-black tree, i.e., (among others) the number of black nodes is the same on each path.

Related Work

Before proceeding further on, let us add a few, more specific comments on the related work in addition to Section 6.1.

Verification of red-black (and other kinds of balanced) trees has, of course, been considered in several works. Out of them, the closest to what we present here is probably the approach of PALE [MS01] (which we briefly characterised already in Section 4.1.1). PALE uses tree automata (although the classical ones only), and it also resembles our work by working with preconditions, postconditions, and loop invariants, and further by reducing the validity problem for Hoare triples to the language emptiness problem. However, PALE as well as most other approaches do not handle the balancedness problem.

In [Par05], preservation of balancedness (together with other safety properties) has been verified for *AVL trees* using TVLA [SRW02] (which we briefly characterised in Section 4.1.2). The author used special, manually provided

instrumentation predicates to track the difference between lengths of branches during rebalancing of AVL trees. This difference can only be between -2 and 2, which can be tracked by five special predicates (that are updated by special, manually provided predicate transformers). In [BCE+05], verification of some properties of *inserting into red-black trees* (including balancedness) is reported. The work uses graph rewriting systems for describing the insertion procedure—the model is manually constructed. Then, an overapproximation by Petri graphs [BCK01] (cf. Section 4.1.5) is used for verifying the fact that two red nodes never appear in succession. Further, graph type systems are used to check the balancedness. Not all desirable safety properties are covered this way, and both of the steps require a significant user involvement.

Recently, [MSZ07] has proposed an approach for verifying algorithms on balanced trees (and, in particular, on red-black trees) based on decidable theories of term algebras with Presburger arithmetic. These theories allow one to define functions from terms to integers, e.g., the maximal number of black nodes in paths from the root to a leave in a tree. The framework of [MSZ07], however, does not allow one to express local updates at an arbitrary control location, which consequently leads to a necessity of using an informal induction when proving program verification conditions. In other recent work [NDQC07], a different formal model—in particular, an extension of separation logic with user-definable shape predicates—is used to reason about safety of pointer-manipulating programs including insertion for red-black trees. This approach also targets the verification of Hoare triples in presence of user-specified program invariants. Unlike in our work, in [NDQC07], the semantics of particular program statements is not handled in an exact way.

Next, let us note that our definition of TASC is the result of searching for a class of counter tree automata that would combine nice closure properties (union, intersection, complementation) with decidability of the emptiness problem. Many automata in the literature concentrate on *in-breadth* counting of nodes (as, e.g., [DL02, SSM03, SSMH04, Lug05]). Our work gives a possibility of *in-depth* counting, which is needed in order to express balancing of recursive tree structures.

It is also worth noticing that various computation models similar to TASC mentioned in the literature, such as *alternating multi-tape and counter automata*[5], have undecidable emptiness problems in the presence of two or more 1-letter input tapes, or, equivalently, in the presence of two non-increasing

[5] Alternating automata generalise the concept of non-deterministic automata by allowing existential and universal transitions. This means that sometimes the rest of the input is required to be accepted from *at least one* of the target states, and sometimes it must be accepted from *all* of the target states. The computation of such a machine may often be conveniently viewed as a tree. Adding alternation to finite-state automata does not increase their expressive power, which is, however, not the case for other types of automata.

counters [Pet95][6] (reading from a 1-letter input tape can be considered as decreasing of a counter encoded in unary). However, restricting the number of counters is problematic for obtaining the closure of automata under intersection as the intersection typically leads to a machine with two counters. The solution we adopt is to let the actions on the counters depend exclusively on the input tree alphabet, this is, we deduce actions to be done on the counters solely from the input. Then, the intersection does not require two counters as both the counters of the original machines change in the same way on the same input. This solution can be seen as a generalisation of the so-called *visibly pushdown languages* [AM04] to trees for singleton stack alphabets. The general case with more than one stack symbol is a subject of future work.

6.2.1 A TASC-based Verification Methodology and a Running Example

In this section, we introduce our verification methodology for programs using balanced trees. In practice, several data structures based on balanced trees are commonly used, e.g., AVL trees. Here, we will use *red-black trees* as our running example. Red-black trees are binary search trees whose nodes are coloured by red or black. They are approximately balanced by constraining the way nodes can be coloured. The constraints insure that no maximal path can be more than twice longer than any other path. Formally, a node contains an element of an ordered data domain, a colour, a left and right pointer, and a pointer to its parent. A *red-black tree* is a binary search tree that satisfies the following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.
4. If a node is red, both its children are black.
5. Each path from the root to a leaf contains the same number of black nodes.

An example of a red-black tree is given in Figure 6.1 (a). The main operations on balanced trees are searching, insertion, and deletion. When implementing the last two operations, one has to make sure that the trees remain balanced. This is usually done using tree rotations—cf. Figure 6.1 (b), which can change the number of black nodes on a given path.

Because of the last condition on red-black trees mentioned above (i.e., having the same number of black nodes in each path), it is obvious that the set of red-black trees is not regular, i.e., not recognisable by standard tree automata [CDG+05]. Therefore, we have to introduce a tree automata model able to describe sets of (heap) configurations containing balanced trees. This model

[6] This result improves on the early work on alternating multi-tape automata recognising 1-letter languages in [Gei91].

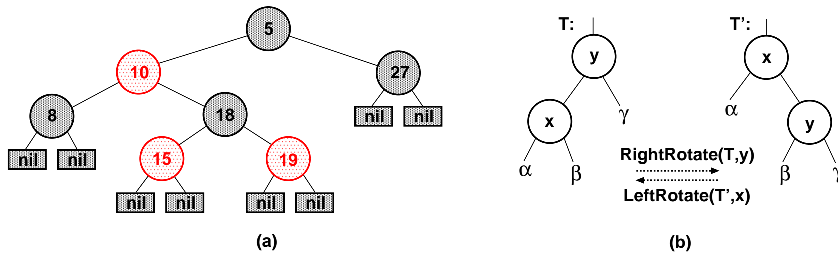


Fig. 6.1. (a) A red-black tree—nodes 10, 15, 19 are red, (b) the left and right tree rotation

has to be powerful enough to describe these trees while still having properties allowing automatic verification (i.e., decidability of inclusion, closure under some operations, etc.).

Here, we define such a class of extended tree automata—namely, tree automata with size constraints (TASC). We suppose the data content of the nodes to be abstracted away—we do not verify sortedness. Basic program blocks (i.e., individual program statements or groups of statements that we view as atomic like, e.g., rotations) define effective transformations on TASC.

We assume the user to specify the precondition and postcondition of the program to be verified. Further, we suppose the user to supply an invariant for each loop. The preconditions and postconditions as well as loop invariants are specified by TASC. Then, the verification is performed by automatically checking correctness of each triple (precondition, program block, postcondition) where the precondition is the program precondition or a loop invariant, the postcondition is the program postcondition or a loop invariant, and the program block is a loop free fragment of the code between the precondition and postcondition. This is done by computing the image of the precondition after an application of the code of the program block and by checking that the image implies the postcondition. This check is done using language inclusion for TASC.

In Fig. 6.2, we give the pseudo-code of the inserting operation for red-black trees [CLR90]. For this program, we want to show that after an insertion of a node, a red-black tree remains a red-black tree. In our work, we restrict ourselves to calculating the effects of program blocks which preserve the tree structure of the heap. This is not the case in general since pointer operations can temporarily break the tree structure, e.g., in the code for performing a rotation. The operations that we handle are the following:

1. tests on the tree structure (like `x->parent == x->parent->parent->left`),
2. changing data of a node (as, e.g., recolouring of a node `x->colour = red`),
3. left and right rotations (Figure 6.1 (b)),

```
RB-Insert(T,x):
  Tree-Insert(T,x);                % Inserts a~new leaf node x
  x->colour = red;
  while (x != root && x->parent->colour == red) {
    if (x->parent == x->parent->parent->left) {
      if (x->parent->parent->right->colour == red) {
        x->parent->colour = black;                  % Case 1
        x->parent->parent->right->colour = black;
        x->parent->parent->colour = red;
        x = x->parent->parent;
      }
      else {
        if (x == x->parent->right) {                % Case 2
          x = x->parent;
          LeftRotate(T,x)
        }
        x->parent->colour = black;                  % Case 3
        x->parent->parent->colour = red;
        RightRotate(T,x->parent->parent);
      }
    }
    else .... % the same as above with right and left exchanged
  }
  root->colour = black;
```

Fig. 6.2. A procedure for inserting into red-black trees

4. moving a pointer up or down a tree structure (like `x = x->parent->parent`),
5. low-level insertion/deletion, i.e., the physical addition/removal of a node to/from a suitable place that is then followed by the re-balancing operations.

6.2.2 The Notion of TASC

In what follows, we work with the set $\mathfrak{D}$ of all boolean combinations of formulae of the form $x - y \diamond c$ or $x \diamond c$ for some $c \in \mathbb{Z}$ and $\diamond \in \{\leq, =, \geq\}$. Notice that negation can be eliminated from any formula of $\mathfrak{D}$ since $x - y \not\leq c \Leftrightarrow x - y \geq c + 1$, and so on. Also, any constraint of the form $x - y \geq c$ can be equivalently written as $y - x \leq -c$. For a closed formula φ, we write $\models \varphi$ meaning that it is valid, i.e., equivalent to true.

Let $Perm(N)$ denote the set of all permutations $I : \{1, \ldots, N\} \to \{1, \ldots, N\}$. The following normal form of formulae from $\mathfrak{D}$ is needed later on in Section 6.2.3.

Lemma 6.2.1 *Every formula $\varphi(x_1, \ldots, x_N)$ of $\mathfrak{D}$ can be effectively written as a disjunction of formulae of the following form, for a suitable permutation $I \in Perm(N)$ of its free variables :*

$$\bigwedge_{k=1}^{N-1} x_{I(k)} - x_{I(k+1)} \diamond_k c_k \;\wedge \bigwedge_{m \in M \subseteq \{1,\ldots,N\}} x_m \leq d_m \;\wedge \bigwedge_{p \in P \subseteq \{1,\ldots,N\}} x_p \geq e_p$$

where $\diamond_k \in \{\leq, =\}$ and $c_k, d_m, e_p \in \mathbb{Z}$.

Proof. First, we eliminate all occurrences of negation and $\geq$. Second, we replace any conjunction of the form $c_1 \leq x - y \leq c_2$ for $c_1 < c_2$ (for $c_1 > c_2$, the conjunction is not satisfiable and the original formula can be simplified accordingly), by the disjunction $\bigvee_{c \in \{c_1, c_1+1, \ldots, c_2\}} x = y + c$. Third, we put the resulting formula in DNF and process each disjunct as follows.

For each permutation $I \in Perm(N)$ of the free variables in φ, we define the induced ordering $\theta_I \;:\; x_{I(1)} \leq x_{I(2)} \leq \ldots \leq x_{I(N)}$. Let $\Theta = \bigvee_{I \in Perm(N)} \theta_I$ be the (logically valid) disjunction of all possible orderings of the free variables $x_1, \ldots, x_N$. In the following, we work with the DNF form of $\varphi \wedge \Theta$, in which each disjunct is necessarily associated with some ordering. We transform each clause (disjunct) $\theta_I \wedge \psi$ of the DNF form of $\varphi \wedge \Theta$ by applying one of the four cases below for each constraint $x_i - x_j \diamond c$, $\diamond \in \{\leq, =\}$, occurring in ψ:

1. If $\theta_I \Rightarrow x_i \leq x_j$ and $c \leq 0$, then there exist $x_i = x_{I(k)} \leq x_{I(k+1)} \leq \ldots \leq x_{I(l)} = x_j$ in θ_I. Let $C = \{\langle c_k, \ldots, c_{l-1}\rangle \mid c_i \geq 0, k \leq i < l,\; \sum_{i=k}^{l-1} c_i = c\}$. Since C is finite, we can replace $x_i - x_j \diamond c$ by the equivalent formula $\bigvee_{\mathbf{c} \in C} \bigwedge_{k \leq i < l} x_{I(i)} - x_{I(i+1)} \diamond c_i$.
2. The case of $\theta_I \Rightarrow x_i \geq x_j$ and $c \geq 0$ is treated in a symmetric way with the first point.
3. If $\theta_I \Rightarrow x_i \leq x_j$ and $c > 0$, the constraint is trivially valid and can be eliminated from the clause. In the case where $x_i - x_j \diamond c$ is the only constraint in the clause, the original formula φ is valid.
4. If $\theta_I \Rightarrow x_i \geq x_j$ and $c < 0$, we discard the entire clause $\theta_I \wedge \psi$ as unsatisfiable. In the case where this was the only clause, the original formula φ is unsatisfiable.

In the resulting formula, we replace:

- any conjunction of constraints of the form $x - y \leq c' \wedge x - y \leq c''$ by $x - y \leq \min(c', c'')$,
- any conjunction of constraints of the form $x - y = c' \wedge x - y = c'$ by simply $x - y = c'$,
- any conjunction of constraints of the form $x - y \leq c' \wedge x - y = c''$ by $x - y = c''$ if $c'' \leq c'$, and
- any conjunction containing a subformula of the form $x - y \diamond c' \wedge x - y = c''$ by $\perp$ if $c' < c''$. □

The size of the disjunction is exponential in the number of variables due to the initial choice over all possible orderings and depends also on the constants c_i due to the choices of the first and fourth numbered item above. Note that this construction does not have to be applied if the number of variables is less than or equal to two, which is our case as we show later on.

As in the case of tree automata in Section 5.1, to be able to define the notion of TASC, we start with a *ranked alphabet* Σ defined as a finite set of symbols together with a rank function $\# : \Sigma \to \mathbb{N}$. For $f \in \Sigma$, the value $\#(f)$ is said to be the *arity* of f. We denote by Σ_n the set of all symbols of arity n from Σ.

Next, let λ denote the empty sequence. A *tree* t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ that satisfies the following conditions:

- $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$, and
- for each $p \in dom(t)$, if $\#(t(p)) = n > 0$, then $\{i \mid pi \in dom(t)\} = \{1, \ldots, n\}$.

A *subtree* of t starting at a position $p \in dom(t)$ is a tree $t_{|p}$ defined as $t_{|p}(q) = t(pq)$ if $pq \in dom(t)$, and undefined otherwise. Given a set of positions $P \subseteq \mathbb{N}^*$, we define the *frontier* of P as the set $fr(P) = \{p \in P \mid \forall i \in \mathbb{N}\ pi \notin P\}$. For a tree t, we use $fr(t)$ as a shortcut for $fr(dom(t))$. We denote $T(\Sigma)$ the set of all trees over the alphabet Σ.

Definition 6.2.1 *Given two trees $t : \mathbb{N}^* \to \Sigma$ and $t' : \mathbb{N}^* \to \Sigma'$, a function $h : dom(t) \to dom(t')$ is said to be a* tree mapping between t and t' *if the following holds:*

- *$h(\lambda) = \lambda$, and*
- *for any $p \in dom(t)$, if $\#(t(p)) = n > 0$, then there exists a prefix-closed set $Q \subseteq \mathbb{N}^*$ such that $pQ \subseteq dom(t')$ and $h(pi) \in fr(pQ)$ for all $1 \le i \le n$.*

A *size function* (or measure) associates to every tree $t \in T(\Sigma)$ an integer $|t| \in \mathbb{Z}$. Size functions are defined inductively on the structure of the tree. For each $f \in \Sigma$, if $\#(f) = 0$, then $|f|$ is a constant c_f, otherwise, for $\#(f) = n$, we have:

$$|f(t_1, \ldots, t_n)| = \begin{cases} b_1|t_1| + c_1 & \text{if } \models \delta_1(|t_1|, \ldots, |t_n|) \\ \quad \ldots & \\ b_n|t_n| + c_n & \text{if } \models \delta_n(|t_1|, \ldots, |t_n|) \end{cases}$$

where $b_1, \ldots, b_n \in \{0, 1\}$, $c_1, \ldots, c_n \in \mathbb{Z}$, and $\delta_1, \ldots, \delta_n \in \mathfrak{D}$, all depending on f. In order to have a consistent definition, it is required that $\delta_1, \ldots, \delta_n$ define a partition of $\mathbb{N}^n$, i.e., $\models \forall x_1 \ldots \forall x_n \ \bigvee_{1 \le i \le n} \delta_i \ \wedge \ \bigwedge_{1 \le i < j \le n} \neg(\delta_i \wedge \delta_j)$.[7] A *sized alphabet* $(\Sigma, |.|)$ is a ranked alphabet with an associated size function.

[7] For technical reasons related to the decidability of the emptiness problem for TASC, we do not allow arbitrary linear combinations of $|t_i|$ in the definition of $|f(t_1, \ldots, t_n)|$.

Definition 6.2.2 *A* tree automaton with size constraints *(TASC) over a sized alphabet* $(\Sigma, |.|)$ *is a 3-tuple* $A = (Q, \Delta, F)$ *where* Q *is a finite set of states,* $F \subseteq Q$ *is a designated set of final states, and* Δ *is a finite set of transition rules of the form* $f(q_1, \ldots, q_n) \xrightarrow{\varphi(|1|, \ldots, |n|)} q$ *where* $f \in \Sigma$*,* $\#(f) = n$*, and* $\varphi \in \mathfrak{D}$ *is a formula with* n *free variables. For constant symbols* $a \in \Sigma$*,* $\#(a) = 0$*, the automaton has unconstrained rules of the form* $a \to q$*.*

A *run* of A over a tree $t : \mathbb{N}^* \to \Sigma$ is a mapping $\pi : dom(t) \to Q$ such that for each position $p \in dom(t)$ where $q = \pi(p)$, we have:

- if $\#(t(p)) = n > 0$ and $q_i = \pi(pi)$, $1 \le i \le n$, then Δ has a rule $t(p)(q_1, \ldots, q_n) \xrightarrow{\varphi(|1|, \ldots, |n|)} q$ and $\models \varphi(|t_{|p1}|, \ldots, |t_{|pn}|)$,
- otherwise, if $\#(t(p)) = 0$, then Δ has a rule $t(p) \to q$.

A run π is said to be *accepting* if and only if $\pi(\lambda) \in F$. As usual, the *language* of A denoted as $\mathcal{L}(A)$ is the set of all trees over which A has an accepting run.

As an example, let us now present a TASC recognising the set of all balanced red-black trees. Let $\Sigma = \{red, black, nil\}$ with $\#(red) = \#(black) = 2$ and $\#(nil) = 0$. First, we define the size function to be the maximal number of black nodes from the root to a leaf: $|nil| = 1$, $|red(t_1, t_2)| = max(|t_1|, |t_2|)$, and $|black(t_1, t_2)| = max(|t_1|, |t_2|) + 1$. The TASC recognising the set of all balanced red-black trees may now be defined as $A_{rb} = (\{q_b, q_r\}, \Delta, \{q_b\})$ with $\Delta = \{nil \to q_b, black(q_{b/r}, q_{b/r}) \xrightarrow{|1| = |2|} q_b, red(q_b, q_b) \xrightarrow{|1| = |2|} q_r\}$. By using $q_{x/y}$ within the left-hand side of a transition rule, we mean the set of two or more rules in which either q_x or q_y take the place of $q_{x/y}$.

6.2.3 Closure and Decidability Properties of TASC

This section is devoted to the closure of the class of TASC under the operations of union, intersection and complement. Decidability of the emptiness problem is also proved here.

Determinisation

A TASC is said to be *deterministic* if, for every input tree, the automaton has at most one run. For every TASC A, we can effectively construct a deterministic TASC A_d such that $\mathcal{L}(A) = \mathcal{L}(A_d)$. We adapt the classical subset construction for determinising bottom-up tree automata. We have to take into account the fact that in a deterministic TASC, two rules which have the same left-hand side should not be applicable simultaneously. This problem is solved below by constructing guards of transition rules of the deterministic TASC as conjunctions of the original transition guards, which could otherwise be in a conflict, and their negations in all possible combinations. This way, we

ensure that all transitions with the same left-hand side have guards that can never be satisfied simultaneously.

Concretely, let $A = (Q, \Delta, F)$. We define $A_d = (Q_d, \Delta_d, F_d)$ where $Q_d = \mathcal{P}(Q)$, $F_d = \{s \in Q_d \mid s \cap F \neq \emptyset\}$, and $f(s_1, \ldots, s_n) \xrightarrow{\varphi} s \in \Delta_d$ if and only if:

$$s \subseteq \{q | f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i\}, \text{ and } s \neq \emptyset$$

$$\varphi = \bigwedge \{\psi | f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i,\ q \in s\} \wedge \bigwedge \{\neg\psi | f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i,\ q \notin s\}$$

In the case of transition rules involving constant symbols, we have $a \rightarrow s \in \Delta_d$ if and only if $s = \{q \mid a \ \rightarrow q \in \Delta\}$. The following theorem proves that non-deterministic and deterministic TASC recognise exactly the same languages.

Theorem 6.2.1 *A_d is deterministic and $\mathcal{L}(A_d) = \mathcal{L}(A)$.*

Proof. The proof is done by induction on the structure of the accepted trees. It is rather technical, and so we skip it here and refer an interested reader to [HIV05]. □

Union, Intersection, and Complement

Let us have two arbitrary TASCs $A_1 = (Q_1, \Delta_1, F_1)$ and $A_2 = (Q_2, \Delta_2, F_2)$. We can assume w.l.o.g. that Q_1 and Q_2 are disjoint. Then, $A_1 \cup A_2 = (Q_1 \cup Q_2, \Delta_1 \cup \Delta_2, F_1 \cup F_2)$. It is easy to check that indeed $\mathcal{L}(A_1 \cup A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. For intersection, let $A_1 \cap A_2 = (Q_1 \times Q_2, \Delta_{12}, F_1 \times F_2)$ where:

$$f((q_1', q_1''), ..., (q_n', q_n'')) \xrightarrow{\varphi' \wedge \varphi''} (q', q'') \in \Delta_{12} \text{ iff } f(q_1', ..., q_n') \xrightarrow{\varphi'} q' \in \Delta_1 \text{ and } f(q_1'', ..., q_n'') \xrightarrow{\varphi''} q'' \in \Delta_2.$$

The fact that $\mathcal{L}(A_1 \cap A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ is again an easy check.

A TASC $A = (Q, \Delta, F)$ is said to be *complete* if for any tree $t \in T(\Sigma)$, there exists a state $q \in Q$ such that $t \xrightarrow[A]{*} q$. An arbitrary TASC can be completed by adding a sink state $\pi \notin Q$ and the following rules for all $f \in \Sigma$, $q_1, \ldots, q_n \in Q$ where $n = \#(f)$:

$$f(q_1, \ldots, q_n) \xrightarrow{\varphi} \pi \in \Delta_c \text{ iff } \varphi = \bigwedge\{\neg\psi \mid f(q_1, \ldots, q_n) \xrightarrow{\psi} q \in \Delta\}$$

$$f(q_1, \ldots, \pi, \ldots q_n) \xrightarrow{\top} \pi \in \Delta_c.$$

Above, Δ_c denotes the set Δ to which the new transition rules have been added. The complete TASC is $A_c = (Q \cup \{\pi\}, \Delta_c, F)$. Notice that if there are no rules $f(q_1, \ldots, q_n) \xrightarrow[A]{\psi} q$, then there is a rule $f(q_1, \ldots, q_n) \xrightarrow[A_c]{\top} q$. It is trivial to check that $\mathcal{L}(A_c) = \mathcal{L}(A)$. Moreover, if A is deterministic, so is A_c.

The *complement* of a deterministic complete TASC $A = (Q, \Delta, F)$ is defined by $\overline{A} = (Q, \Delta, Q \setminus F)$. The proof that $\mathcal{L}(\overline{A}) = T(\Sigma) \setminus \mathcal{L}(A)$ is as in the case of classical tree automata [CDG+05].

Emptiness

In this subsection, we give an effective method for deciding emptiness of the language of a TASC. In fact, we address a slightly more general problem—given a TASC $A = (Q, \Delta, F)$, we construct for each state $q \in Q$, an arithmetic formula $\phi_q(x)$ in one variable that precisely characterises the sizes of the trees whose roots are labelled with q by A, i.e., $\models \phi_q(n)$ iff $\exists t.|t| = n$ and $t \xrightarrow[A]{*} q$. As it will turn out, the ϕ_q formulae are expressible in Presburger arithmetic, therefore their satisfiability is decidable [Pre29]. This entails the decidability of the emptiness problem for TASC, which can be expressed as the satisfiability of the disjunction $\bigvee_{q \in F} \phi_q$.

In order to construct ϕ_q, we shall translate our TASC into an alternating pushdown system (APDS) whose stack encodes the value of one integer counter, denoted by y from now on. An APDS is a triple $S = (Q, \Gamma, \delta, F)$ where Q is a finite set of control locations, Γ is a finite stack alphabet, $F \subseteq Q$ is a set of final control locations, and δ is a mapping from $Q \times \Gamma$ into $\mathcal{P}(\mathcal{P}(Q \times \Gamma^*))$. Notice that an APDS does not have an input alphabet since we are interested in the behaviours it generates, rather than in the accepted language. A run of an APDS is a tree $t : \mathbb{N}^* \to (Q \times \Gamma^*)$ satisfying the following property: for any $p \in dom(t)$, if $t(p) = \langle q, \gamma w \rangle$, then $\{t(pi) \mid 1 \le i \le \#(t(p))\} = \{\langle q_1, w_1 w \rangle, \ldots, \langle q_n, w_n w \rangle\}$, where $\{\langle q_1, w_1 \rangle, \ldots, \langle q_n, w_n \rangle\} \in \delta(q, \gamma)$. The run is accepting if all control locations occurring on its frontier are final.

Once we have an APDS, we use the construction of [BEM97] to calculate the set $\mathrm{pre}_q^*(\sigma)$ of configurations c with a control state q that have a successor set in a given set of configurations σ, i.e., $c = \langle q, w \rangle \xrightarrow{*} C \subseteq \sigma$. It is shown in [BEM97] that if σ is a regular language, then so is $\mathrm{pre}^*(\sigma)$, and the alternating finite automaton recognising the latter can be constructed in time polynomial in the size of the APDS. Hence, the Parikh images of such $\mathrm{pre}_q^*(\sigma)$ sets are semilinear sets definable by Presburger formulae. In our case, $\sigma = \{\langle q, \epsilon \rangle \mid q \in F\}$ is a finite set where ϵ is the (encoding of the) empty stack. Using a unary encoding of the counter in a stack, we obtain the needed formulae $\phi_q(x)$.

Given a TASC $A = (Q, \Delta, F)$ over an alphabet $(\Sigma, |.|)$, let $S_A = (Q_A, \Gamma, \delta_A, F_A)$ be the APDS where $Q_A = (Q \times \Sigma) \cup \Pi$, $\Gamma = \{-, 0, 1\}$, and $F_A = \{q_f\} \subset \Pi$. Here, Π is an additional set of states that are needed in

the construction of S_A from A and that are not of the form $\langle q, f\rangle$. We use 0 as the beginning of the stack marker, $-$ on the top of the stack denotes a negative value, and 1 is used for the unary encoding of the absolute value of the counter. We shall represent an integer counter x by a stack configuration 1^n0 if the value of x is $n \in \mathbb{N}$, and -1^n0 if its value is $-n$. The primitive operations on x, i.e., the increment, decrement, and zero test are encoded by the moves given in Figure 6.3.

$q \xrightarrow{x'=x+1} q'$	$q \xrightarrow{x'=x-1} q'$	$q \xrightarrow{x=0} q'$
$\langle q,1\rangle \hookrightarrow \langle q',11\rangle$ $\langle q,0\rangle \hookrightarrow \langle q',10\rangle$ $\langle q,-\rangle \hookrightarrow \langle q^-,\epsilon\rangle$ $\langle q^-,1\rangle \hookrightarrow \langle q'^-,\epsilon\rangle$ $\langle q'^-,1\rangle \hookrightarrow \langle q',-1\rangle$ $\langle q'^-,0\rangle \hookrightarrow \langle q',0\rangle$	$\langle q,1\rangle \hookrightarrow \langle q',\epsilon\rangle$ $\langle q,0\rangle \hookrightarrow \langle q',-10\rangle$ $\langle q,-\rangle \hookrightarrow \langle q',-1\rangle$	$\langle q,0\rangle \hookrightarrow \langle q',0\rangle$

Fig. 6.3. Encoding a counter by a stack

We shall encode a move of A as a series of moves of S_A. As A moves bottom-up on the tree, S_A will perform a series of alternating top-down transitions, simulating the move of A in reverse. The stack (counter) of S_A is intended to encode the value of the size function $|.|$ at the current tree node.

Suppose that A has a transition rule $f(q_1,\dots,q_n) \xrightarrow{\varphi} q$ and that the current node is of the form $f(t_1,\dots,t_n)$ with $|f(t_1,\dots,t_n)| = b_r|t_r| + c_r$, and δ_r is the disjunctive condition such that $\models \delta_r(|t_1|,\dots,|t_n|)$, according to the definition of the size function (see Section 6.2.2). W.l.o.g. we consider from now on that φ and δ_r have the same set of free variables, denoted $x_1,\dots,x_n$. In what follows, we consider the case $b_r = 1$, i.e. $|f(t_1,\dots,t_n)| = |t_r| + c_r$. The case $b_r = 0$ can be treated in a similar way, by guessing the value $|t_r|$. The position r is said to be the *reference position* of the subtree $f(t_1,\dots,t_n)$. The value $|t_r|$ is said to be the *reference value* of $f(t_1,\dots,t_n)$.

Without losing generality, we consider that the difference constraint formula $\varphi \wedge \delta_r \in \mathfrak{D}$ has already been converted into the normal form of Lemma 6.2.1, that is, a disjunction of formulae of the form:

$$\bigwedge_{k=1}^{n-1} x_{I(k)} - x_{I(k+1)} \diamond_k d_k \;\wedge \bigwedge_{m\in M\subseteq\{1,\dots,n\}} x_m \le e_m \;\wedge \bigwedge_{p\in P\subseteq\{1,\dots,n\}} x_p \ge l_p$$

where $\diamond_k \in \{\le, =\}$, $d_k, e_m, l_p \in \mathbb{Z}$, and $I \in \mathit{Perm}(n)$. For the rest of this section, let us fix one such disjunct.

After each sequence of universal moves, S_A creates n copies of its counter y, let us name them $y_1,\dots,y_n$. The counter y_i is intended to hold the value $|t_{I(i)}|$ for $1 \le i \le n$, and the counter y holds the value $|f(t_1,\dots,t_n)|$. Let

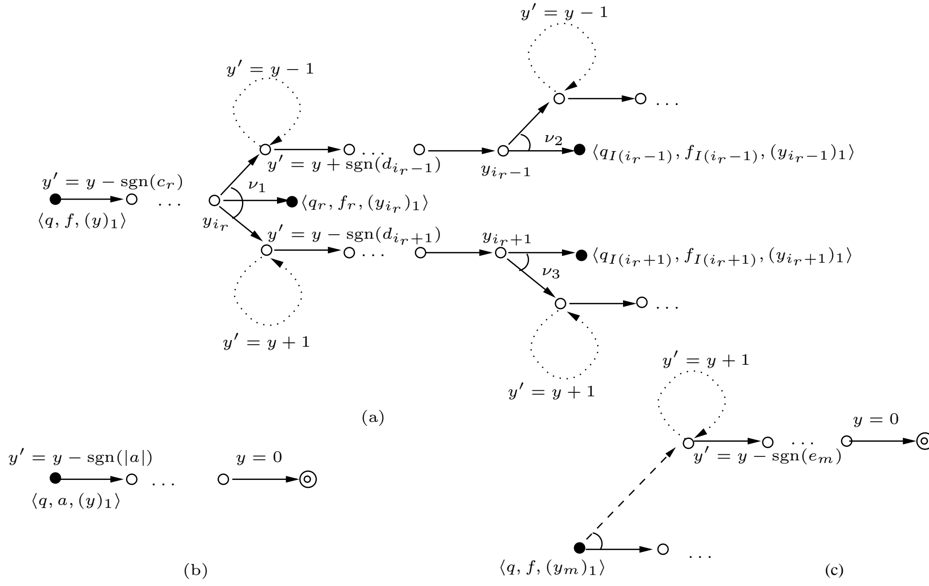


Fig. 6.4. Simulation of a TASC by an APDS

$i_r = I^{-1}(r)$ be the index of the counter y_{i_r} that holds the reference value of the given transition, i.e. $y = y_{i_r} + c_r$. With this notation, Figure 6.4 (a) shows the alternating moves of S_A that simulate the A-transition considered, for one disjunct of $\varphi \wedge \delta_r$. Figure 6.4 (b) shows the moves for transitions of the form $a \rightarrow q$.

Filled circles in Figure 6.4 represent states from $Q \times \Sigma$, and empty circles are additional states from Π. The only accepting state of S_A, named q_f, is marked by a double circle. The notation sgn(...) denotes the sign function, i.e. $\mathrm{sgn}(n) = 1$ if $n > 0$, $\mathrm{sgn}(0) = 0$, and $\mathrm{sgn}(n) = -1$ if $n < 0$. Next, $\nu_1, \nu_2, \ldots$ are symbolic names for the universal moves performed by S_A. Further, in what follows, we will denote a configuration $\langle\langle q, f\rangle, u\rangle$ of S_A by writing $\langle q, f, u\rangle$. In particular, in Figure 6.4, configurations from $Q \times \Sigma \times \Gamma^*$ are labelled by triples of the form $\langle q, f, (y)_1\rangle$. Here, $(y)_1$ denotes the unary encoding of the value of the y counter. Moreover, for simplicity, configurations from $\Pi \times \Gamma^*$ are labelled only with $(y)_1$ in Figure 6.4.

When simulating the A-transition $f(q_1, \ldots, q_n) \xrightarrow{\varphi} q$, S_A starts with the configuration $\langle q, f, (y)_1\rangle$ (cf. Figure 6.4 (a)). In order to derive the reference value y_{i_r} from y, S_A performs $|c_r|$ decrement or increment actions, depending on whether the sign of c_r is positive or negative. Then S_A performs the universal move ν_1 making three copies of itself (unless $i_r = 1$ when the upper branch is omitted and/or $i_r = n$ when the lower branch is omitted). The middle branch simply moves to the appropriate control state $\langle q_r, f_r\rangle$ with stack

$(y_{i_r})_1$. The upper and lower branches are used to produce the values y_{i_r-1} and y_{i_r+1} if needed.

The upper branch of the universal move ν_1 depicted in Figure 6.4 depends on $\diamond_r \in \{\leq, =\}$. If $\diamond_r$ is $=$, then S_A performs a sequence of increment/decrement operations of length d_{i_r-1} in order to obtain the value y_{i_r-1} from y_{i_r} (since $y_{i_r-1} = y_{i_r} + d_{i_r-1}$). If $\diamond_r$ is $\leq$, then there is an additional existential (non-deterministic) transition—depicted using a dotted line in Figure 6.4 (a)—which decrements the counter an arbitrary number of times in order to obtain a smaller value (since $y_{i_r-1} \leq y_{i_r} + d_{i_r-1}$).

A similar sequence of transitions is performed by the lower branch of ν_1. Note that the symbols $f_{I(i_r-1)}, f_r, f_{I(i_r+1)}$ are chosen arbitrarily, that is, for each triple $(g_1, g_2, g_3) \in \Sigma_n^3$, S_A performs three universal moves that are identical to ν_1, ν_2, ν_3, with g_1, g_2, and g_3 substituted for $f_{I(i_r-1)}, f_r$, and $f_{I(i_r+1)}$, respectively.

Next, if $i_r - 1 > 1$, the simulation continues with the binary universal move ν_2. The lower branch of ν_2 changes the control into $\langle q_{I(i_r-1)}, f_{I(i_r-1)}\rangle$ without changing the stack. The upper branch of ν_2 leads to a control state from Π, from which the remaining values $y_{i_r-2}, \ldots, y_1$ are produced. Symmetrically, the universal move ν_3 leads to configurations producing the values $y_{i_r+1}, \ldots, y_n$.

Clearly, the values of the counters $y_1, y_2, \ldots, y_n$ that are obtained in the way described above will satisfy the constraint $\varphi \wedge \delta_r$ when used as the sizes of the subtrees $t_{I(1)}, t_{I(2)}, ..., t_r, ..., t_{I(n)}$. Moreover, at the same time, any assignment satisfying this formula can be obtained in some run of S_A by iterating the increment/decrement self-loops a sufficient number of times.[8]

In order to simulate moves of the form $a \to q$ (Figure 6.4 (b)), S_A simply decrements/increments the counter, depending on the sign of $|a|$, a number of times equal to the absolute value of $|a|$. The condition $y = 0$ ensures that S_A accepts only with the empty stack. The universal dotted branch in Figure 6.4 (c) is used to test that $y_m \leq e_m$ for some $1 \leq m \leq n$. A similar test for $y_p \geq l_p$ can be issued by replacing $y' = y + 1$ with $y' = y - 1$ on the loop. The following lemma is a concretisation of the above considerations:

Lemma 6.2.2 *Let $A = (Q, \Delta, F)$ be a TASC over a sized alphabet $(\Sigma, |.|)$ and let S_A be its corresponding APDS.*

1. *For any tree $t \in T(\Sigma)$ and any run $\pi : dom(t) \to Q$ of A on t, there exists an accepting run $\rho : \mathbb{N}^* \to (Q \times \Sigma \cup \Pi) \times \Gamma^*$ of S_A and an injective tree mapping $h : dom(t) \to dom(\rho)$ between π and ρ such that:*

$$\forall p \in dom(t) \ . \ \rho(h(p)) = \langle \pi(p), t(p), (|t_{|p}|)_1 \rangle \tag{6.1}$$

[8] Notice that since APDS do not have input, the universal branches are not synchronised, hence the iterations can be performed separately.

2. *For any accepting run* $\rho : \mathbb{N}^* \rightarrow (Q \times \Sigma \cup \Pi) \times \Gamma^*$ *of* S_A*, there exists a tree* $t \in T(\Sigma)$*, a run* $\pi : dom(t) \rightarrow Q$ *of* A *on* t*, and an injective tree mapping* $h : dom(t) \rightarrow dom(\rho)$ *between* π *and* ρ *satisfying (6.1).*

Proof. The proof is rather long and quite technical—as we gave the intuition behind the construction before stating the lemma, we skip the exact proof here and refer an interested reader to [HIV05]. □

We can now formalise the main result of this subsection.

Theorem 6.2.2 *Let* A *be a TASC. The problem whether* $L(A) = \emptyset$ *is decidable.*

Proof. Due to Lemma 6.2.2, we know that a tree with a root symbol $f \in \Sigma$ is accepted at a state q of a TASC $A = (Q, \Delta, F)$ over a sized alphabet Σ iff there is an accepting run from the control state $\langle q, f \rangle$ in the appropriate APDS $S_A = (Q_A, \Gamma, \delta_A, F_A)$. It is thus enough to use the result of [BEM97] (mentioned at the beginning of the section) to check whether for some $\langle q, f \rangle \in Q_A$ where $q \in F$, $pre^*_{\langle q,f \rangle}(\{\langle q_{fin}, \varepsilon \rangle\})$ is non-empty. Here, q_{fin} is the unique final state of the APDS S_A constructed according to Figure 6.4. □

As a remark, let us note that the decidability of the emptiness problem for TASC can also be proved via a reduction to the class of *tree automata with one memory* [CC05] by encoding the size of a tree as a unary term. The inequality constraints from the guards of the TASC can be simulated analogously by adding increment/decrement self-loops to the tree automata with one memory.

6.2.4 Semantics of Tree Updates

As explained in Section 6.2.1, there are three types of operations that commonly appear in procedures used for balancing binary trees after an insertion or deletion: (1) navigation in a tree, i.e., testing or changing the position a pointer variable is pointing to in the tree, (2) testing or changing certain data fields of the encountered tree nodes, such as the colour of a node in a red-black tree, and (3) tree rotations. In addition, one has to consider the physical insertion or deletion to/from a suitable position in the tree as an input for the re-balancing.

It turns out that the TASC defined in Section 6.2.2 are not closed with respect to the effect of some of the above operations, in particular the ones that change the balance of subtrees (the difference between the size of the left and right subtree at a given position in the tree). Therefore, we now introduce a subclass of TASC called *restricted TASC* (rTASC) which we show to be closed with respect to all the needed operations on balanced trees. Moreover,

rTASC are closed with respect to intersection and union, amenable to determinisation and minimisation, though not closed with respect to complementation. The idea is to use rTASC to express loop invariants and preconditions and postconditions of programs as well as to perform the necessary reachability computations. TASC are then used in the associated language inclusion checks (where they arise via negation of rTASC).

Restricted TASC

A *restricted alphabet* is a sized alphabet consisting only of nullary and binary symbols and a size function of the form $|f(t_1, t_2)| = \max(|t_1|, |t_2|) + a$ with $a \in \mathbb{Z}$ for binary symbols. A *restricted* TASC is a TASC with a restricted alphabet and with binary rules only of the form $f(q_1, q_2) \xrightarrow{|1|-|2|=b} q$ with $b \in \mathbb{Z}$.

Notice that any conjunction of guards of an rTASC and their negations reduces either to false, or to only one formula of the same form, namely $|1| - |2| = b$. Using this fact, one can show that the intersection of two rTASC is again an rTASC, and that applying the determinisation of Section 6.2.3 to an rTASC yields another rTASC. Moreover, the intersection of an rTASC with a classical tree automaton is again an rTASC.[9] Clearly, rTASC are not closed under complementation as inequality guards are not allowed.

Minimisation of rTASC

The simple form of the guards allows us to have a practical minimisation procedure based on the minimisation for classical bottom-up tree automata [CDG$^+$05]. If $(\Sigma, |.|)$ is a restricted alphabet, let Σ_δ be the infinite ranked alphabet $\{\langle f, d\rangle \mid f \in \Sigma, d \in \mathbb{Z}\}$ with $\#(\langle f, d\rangle) = \#(f)$. For any $t \in T(\Sigma)$, let $\delta(t) \in T(\Sigma_\delta)$ be the tree defined by the following conditions:

- $dom(t) = dom(\delta(t))$,
- for all $p \in dom(t)$, if $\#(t(p)) = 0$, we have $\delta(t)(p) = \langle t(p), |t(p)|\rangle$, and
- for all $p \in dom(t)$, if $\#(t(p)) = 2$, we have $\delta(t)(p) = \langle t(p), |t_{|p1}| - |t_{|p2}|\rangle$.

Obviously, δ is a (bijective) function from $T(\Sigma)$ to $T(\Sigma_\delta)$, which we extend point-wise to sets of trees. If A is an rTASC over the restricted alphabet $(\Sigma, |.|)$, let A_δ be the bottom-up tree automaton over Σ_δ defined by replacing each transition rule of A of the form:

- $a \to q$ by $\langle a, |a|\rangle \to q$, and
- $f(q_1, q_2) \xrightarrow{|1|-|2|=b} q$ by $\langle f, b\rangle(q_1, q_2) \to q$.

[9] A bottom-up tree automaton can be seen as a TASC in which all guards are true.

Note that we can always define A_δ over a finite subset of Σ_δ since the number of rules in A is finite. Moreover, the size of A (number of states) equals the size of A_δ. Last, the transformation of A into A_δ is always reversible.

Lemma 6.2.3 *Given an rTASC A over a sized alphabet $(\Sigma, |.|)$, for all trees $t \in T(\Sigma)$, we have $t \in \mathcal{L}(A)$ if and only if $\delta(t) \in \mathcal{L}(A_\delta)$.*

Proof. We prove that $t \xrightarrow[A]{*} q$ iff $\delta(t) \xrightarrow[A_\delta]{*} q$ by induction on the structure of t. If $t = a \in \Sigma_0$, $a \xrightarrow[A]{} q$ if and only if $\delta(a) = \langle a, |a| \rangle \xrightarrow[A_\delta]{} q$. Otherwise, let $t = f(t_1, t_2) \xrightarrow[A]{*} f(q_1, q_2) \xrightarrow[A]{|1| - |2| = b} q$ with $t_i \xrightarrow[A]{*} q_i$, $1 \leq i \leq 2$. Then, $|t_1| - |t_2| = b$, and hence $\delta(t) = \langle f, b \rangle(\delta(t_1), \delta(t_2))$. By the induction hypothesis, we have $\delta(t_i) \xrightarrow[A_\delta]{*} q_i$ and, by the definition of A_δ, $\langle f, b \rangle(q_1, q_2) \xrightarrow[A_\delta]{} q$. The other direction is symmetrical. □

Now, given an rTASC A, we compute A_δ, minimise it using the classical construction from [CDG+05] obtaining ${A_\delta}^{min}$. The minimal rTASC A^{min} is obtained by performing the reverse operation on ${A_\delta}^{min}$, i.e., moving back the integer constants from the symbols to the guards. To convince ourselves that A^{min} is indeed minimal, suppose there exists a smaller rTASC A' recognising the same language, i.e., $\mathcal{L}(A) = \mathcal{L}(A^{min}) = \mathcal{L}(A')$. Then, $\delta(\mathcal{L}(A)) = \delta(\mathcal{L}(A')) = \mathcal{L}(A'_\delta) = \mathcal{L}({A_\delta}^{min})$. Since A' and A'_δ have the same number of states, we contradict the minimality of ${A_\delta}^{min}$.

6.2.5 Representing Sets of Memory Configurations

To be able to describe how tree rotations (and the other considered operations) can be implemented over rTASC, we first have to explain how rTASC can be used for describing sets of memory configurations of programs manipulating balanced tree structures like red-black trees or AVL trees. Note that the memory representation used here is much simpler than the one from [BHRV06b] mentioned in Section 5.4. This is because we work with tree-shaped heaps only and thus we can map heap graphs directly onto the trees accepted by rTASC with nodes labelled by the variables pointing to them and by the data elements stored in them. We also use the label *nil* to denote null successors of leaf nodes.

Formally, let us consider a finite set of *pointer variables* $\mathcal{V} = \{x, y, \ldots\}$ and a finite set of data values $\mathcal{D}$ disjoint with $\mathcal{V}$, e.g., $\mathcal{D} = \{red, black\}$. In the following, we let $\Sigma = \mathcal{P}(\mathcal{V} \cup \mathcal{D} \cup \{nil\})$ where *nil* indicates a null pointer value ($nil \notin \mathcal{V} \cup \mathcal{D}$). The arity function is defined as follows: $\#(f) = 2$ if $nil \notin f$, and $\#(f) = 0$ otherwise. For a tree $t \in T(\Sigma)$ and a variable $x \in \mathcal{V}$, we say that a position $p \in dom(t)$ is *pointed to by* x if and only if $x \in t(p)$.

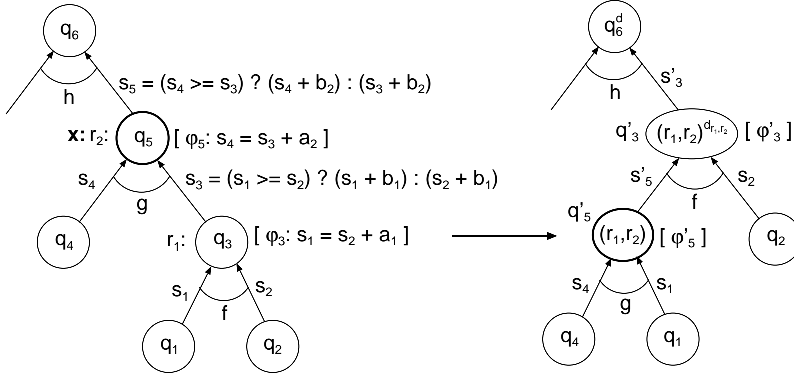


Fig. 6.5. Left rotation on an rTASC

For the rest of the section, let $A = (Q, \Delta, F)$ be an rTASC over Σ. We say that A represents a *set of memory configurations* if and only if for each $t \in L(A)$ and each $x \in \mathcal{V}$, there is at most one $p \in dom(t)$ such that $x \in t(p)$. This condition can be ensured by the construction of A where we let $Q = \mathcal{Q} \times \mathcal{P}(\mathcal{V})$ for some finite set $\mathcal{Q}$ and we let Δ consist only of rules of the form $f(\langle q_1, v_1\rangle, \langle q_2, v_2\rangle) \xrightarrow{\varphi} \langle q, v\rangle$ where (1) $v = (f \cup v_1 \cup v_2) \cap \mathcal{V}$ and (2) $f \cap v_1 = f \cap v_2 = v_1 \cap v_2 = \emptyset$. Intuitively, a control state $\langle q, v\rangle$ "remembers" all variables encountered by condition (1), while condition (2) ensures that no variable is encountered twice.

Remark. To simplify the presentation of the effect of program statements on a set of memory configurations given by an rTASC, we suppose in the following that the statements do not lead to a memory error (like a null pointer dereference or similar). However, it is easy to implement tests for these potential errors over sets of memory configurations described by rTASCs in the same way as regular program conditions (i.e. if statements) are implemented, which we explain in Section 6.2.5.

Modelling Tree Rotations

Let $x \in \mathcal{V}$ be a fixed variable. We shall construct an rTASC $A' = (Q', \Delta', F')$ that describes the set of trees that are the result of the *left rotation* of a tree from $L(A)$ applied at the node pointed to by x. The case of the right tree rotation is very similar.[10] In the description, we will be referring to Figure 6.5 illustrating the problem.

[10] In fact, it can be implemented by temporarily swapping the child nodes in the involved rules, doing a left rotation, and then swapping the child nodes again.

Let $R_x = \{(r_1, r_2) \in \Delta^2 \mid x \in g \wedge r_1 : f(q_1, q_2) \xrightarrow{\varphi_3} q_3 \wedge r_2 : g(q_4, q_3) \xrightarrow{\varphi_5} q_5\}$ be the set of all the pairs of automata rules that can yield a rotation and be modified because of it. Other rules may then have to be modified to reflect the change in one of their *left hand side states*, e.g., the change of q_5 to q_3' in the h-rule in Figure 6.5, or to reflect the *change in the balance* that may result from the rotation, i.e., a change in the *difference of the sizes* of the subtrees of some node. We discuss later what changes in the balance can appear after a rotation, and Lemma 6.2.4 proves that the set D of the possible changes in the balance in the described trees is finite. The automaton A' can thus be constructed from A as follows:

1. $Q' = Q \cup R_x \cup (R_x \times D) \cup (Q \times D)$ where we add new states for the rotated parts and to reflect the changes in the balance.
2. $\Delta' = \Delta \cup \Delta_r \cup \beta(\Delta \cup \Delta_a)$ where:
   - Δ_r corresponding to the rotated rules is the smallest set such that for all $(r_1, r_2) \in R_x$ where $r_1 : f(q_1, q_2) \xrightarrow{\varphi_3} q_3$ and $r_2 : g(q_4, q_3) \xrightarrow{\varphi_5} q_5$, Δ_r contains the rules $g(q_4, q_1) \xrightarrow{\varphi_5'} q_5'$ and $f(q_5', q_2) \xrightarrow{\varphi_3'} q_3'$ where $q_5' = (r_1, r_2)$ and $q_3' = (r_1, r_2)^{d_{r_1,r_2}}$. Here, we use $(r_1, r_2)^{d_{r_1,r_2}}$ as a shorthand for $\langle (r_1, r_2), d_{r_1,r_2} \rangle$. The value $d_{r_1,r_2} \in \mathbb{Z}$ represents the change in the balance caused by the rotation based on r_1, r_2. We describe the computation of φ_3', φ_5', and d_{r_1,r_2} below.
   - Δ_a is the set of rules that could be applied just above the position where a rotation takes place. For each $(r_1, r_2) \in R_x$, we take all rules from Δ that have q_5 within the left hand side and add them to Δ_a with (r_1, r_2) substituted for q_5.
   - β (described in detail in Section 6.2.5) is the function that implements the necessary changes in the guards and input/output states (adding the d-component) of the rules due to the changes in the balance.
3. $F' = (F \times D) \cup F_r$. Here, F_r captures the case where q_3' becomes accepting, i.e., the right child of the node previously labelled by q_3 becomes the root of the entire tree.

Suppose that φ_3 is $|t_1| = |t_2| + a_1$, and let us denote the sizes of the subtrees read at q_1 and q_2 before the rotation by s_1 and s_2, respectively. Let the size function associated with f be $|f(t_1, t_2)| = \max(|t_1|, |t_2|) + b_1$, and let s_3 denote the size of the subtree labelled by q_3 before the rotation. Also, suppose that φ_5 is $|t_1| = |t_2| + a_2$, and let us denote the size of the sub-tree read at q_4 before the rotation as s_4. Finally, let the size function associated with g be $|g(t_1, t_2)| = \max(|t_1|, |t_2|) + b_2$, and let s_5 denote the size of the subtree labelled by q_5 before the rotation. We denote s_5' and s_3' the sizes obtained at q_5' and q_3' after the rotation.

The key observation that allows us to compute φ_3', φ_5', and d_{r_1,r_2} is that due to the chosen form of guards and sizes, we can always compute any two of the sizes s_1, s_2, s_4 from the remaining one. Indeed,

- for $a_1 \geq 0$, we have $s_3 = s_1 + b_1 = s_2 + a_1 + b_1 = s_4 - a_2$, whereas
- for $a_1 < 0$, we have $s_3 = s_2 + b_1 = s_1 - a_1 + b_1 = s_4 - a_2$.

Computing φ'_3, φ'_5, and d_{r_1,r_2} is then just a complex exercise in case splitting. Notice that all the cases can be distinguished statically according to the mutual relations of the constants a_1, b_1, a_2, and b_2. In the case of φ'_5, we obtain the following:

1. For $a_1 \geq 0$, we have $s_4 = s_1 + b_1 + a_2$, and so φ'_5 relating a subtree of size s_4 and s_1 (cf. Figure 6.5) is $|t_1| = |t_2| + b_1 + a_2$.
2. For $a_1 < 0$, we have $s_4 = s_1 - a_1 + b_1 + a_2$, and so φ'_5 is $|t_1| = |t_2| - a_1 + b_1 + a_2$.

The guard φ'_3 is a bit more complex. We distinguish two cases—$\Phi_{4\geq 1} : s_4 \geq s_1$ and $\Phi_{4<1} : s_4 < s_1$. Now, we rewrite the conditions $s_4 \geq s_1$ and $s_4 < s_1$ using the relation between s_4 and s_1 described above for $a_1 \geq 0$ and $a_1 < 0$:

1. $\Phi_{4\geq 1} : s_4 \geq s_1 \iff (a_1 \geq 0 \wedge b_1 + a_2 \geq 0) \vee (a_1 < 0 \wedge -a_1 + b_1 + a_2 \geq 0)$. If $\Phi_{4\geq 1}$ holds, then $s'_5 = s_4 + b_2$. Further, we distinguish between the following cases:
   a) For $a_1 \geq 0 \wedge b_1 + a_2 \geq 0$, we get $s'_5 = s_1 + b_1 + a_2 + b_2$ (as $a_1 \geq 0$), i.e., $s_1 = s'_5 - b_1 - a_2 - b_2$. Taking into account that $s_1 = s_2 + a_1$, we obtain $\varphi'_3 : |t_1| = |t_2| + a_1 + b_1 + a_2 + b_2$.
   b) For $a_1 < 0 \wedge -a_1 + b_1 + a_2 \geq 0$, we have $s'_5 = s_1 - a_1 + b_1 + a_2 + b_2$ (as $a_1 < 0$), i.e., $s_1 = s'_5 + a_1 - b_1 - a_2 - b_2$. Using that $s_1 = s_2 + a_1$, we obtain $\varphi'_3 : |t_1| = |t_2| + b_1 + a_2 + b_2$.
2. $\Phi_{4<1} : s_4 < s_1 \iff (a_1 \geq 0 \wedge b_1 + a_2 < 0) \vee (a_1 < 0 \wedge -a_1 + b_1 + a_2 < 0)$. If $\Phi_{4<1}$ holds, we have $s'_5 = s_1 + b_2$, and so $\varphi'_3 : |t_1| = |t_2| + a_1 + b_2$.

The computation of the change in the balance d_{r_1,r_2} is similar to the above. The first case to be considered is $\Phi_{4\geq 3} : s_4 \geq s_3 \iff a_2 \geq 0$. Here, $s_5 = s_4 + b_2$. To compute the change in the sizes reached at q_5 and q'_3, which is to be compensated in the transitions to come after q'_3 instead of q_5, we need to compute s'_3 as a function of s_4 (then, in the difference, s_4 will be eliminated). We can write the following:

$$s'_3 = \begin{cases} \text{if } \Phi_{4\geq 1} : \\ \quad \begin{cases} \text{if } s_4 + b_2 \geq s_2 : s_4 + b_2 + b_1 \\ \text{if } s_4 + b_2 < s_2 : s_2 + b_1 \end{cases} \\ \text{if } \Phi_{4<1} : \\ \quad \begin{cases} \text{if } s_1 + b_2 \geq s_2 : s_1 + b_2 + b_1 \\ \text{if } s_1 + b_2 < s2 : s_2 + b_1 \end{cases} \end{cases}$$

Let us first consider the subcase when $\Phi_{4\geq 1}$. It has two further subcases $s_4 + b_2 \geq s_2$ and $s_4 + b_2 < s_2$, which we can again rewrite by using the known relations between s_4 and s_2 for $a_1 \geq 0$ ($s_2 + a_1 + b_1 = s_4 - a_2$) and $a_1 < 0$ ($s_2 + b_1 = s_4 - a_2$). We get:

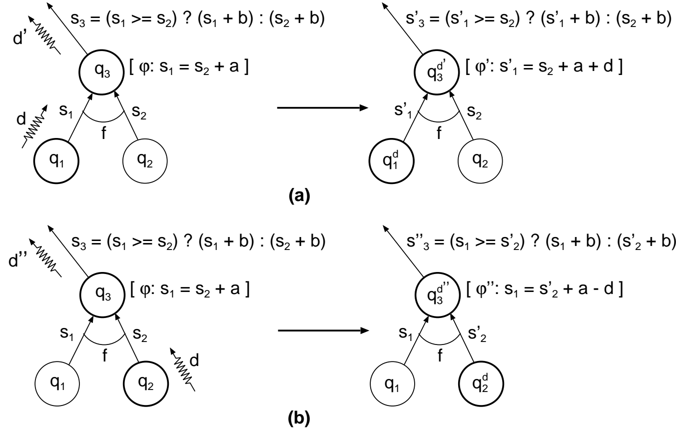


Fig. 6.6. Propagation of changes in the balance in an rTASC

1. $s_4+b_2 \geq s_2 \iff (a_1 \geq 0 \wedge a_1+b_1+a_2+b_2 \geq 0) \vee (a_1 < 0 \wedge b_1+a_2+b_2 \geq 0)$. In this case, we have $s'_3 = s_4 + b_2 + b_1$, and so $d_{r_1,r_2} = b_1$.
2. $s_4+b_2 < s_2 \iff (a_1 \geq 0 \wedge a_1+b_1+a_2+b_2 < 0) \vee (a_1 < 0 \wedge b_1+a_2+b_2 < 0)$. Here, $s'_3 = s_2 + b_1$, and we distinguish the following subcases:
   a) For $a_1 \geq 0 \wedge a_1+b_1+a_2+b_2 < 0$, $s'_3 = s_2+b_1 = s_4-a_1-b_1-a_2+b_1 = s_4 - a_1 - a_2$, and so $d_{r_1,r_2} = -a_1 - a_2 - b_2$.
   b) For $a_1 < 0 \wedge b_1+a_2+b_2 < 0$, $s'_3 = s_2+b_1 = s_4-b_1-a_2+b_1 = s_4-a_2$, and so $d_{r_1,r_2} = -a_2 - b_2$.

The remaining cases of the d_{r_1,r_2} computation are similar to the above.

Propagating Changes in the Balance through rTASC

As we have already said above, tree updates such as recolouring or rotations may introduce changes in the balance at certain points. These changes may affect the balance at all positions above the considered node. The role of the β function is to propagate a change d in the balance upwards in the trees recognised by the given rTASC. The way β changes a set of rules is illustrated in Figure 6.6. For every $d \in D$, every input rule $f(q_1, q_2) \xrightarrow{\varphi} q_3$ is changed to two rules $f(q_1^d, q_2) \xrightarrow{\varphi'} q_3^{d'}$ and $f(q_1, q_2^d) \xrightarrow{\varphi''} q_3^{d''}$ corresponding to the cases when the change in the balance originates from the left or the right. Since we consider just one rotation in every tree (at a given node pointed to by the pointer variable x), the change can never come from both sides. The new guards are $\varphi' : |t_1| = |t_2| + a + d$ and $\varphi'' : |t_1| = |t_2| + a - d$. Let us further analyse the changes in the balance propagated upwards after d comes from the bottom.

Suppose the change in balance is coming from the left as in Figure 6.6 (a). We distinguish the cases of $a \geq 0$ and $a < 0$. (1) For $a \geq 0$, the original size at q_3 is $s_3 = s_1 + b$ where s_1 is the original size at q_1. After the change d happens at q_1, i.e., $s'_1 - s_1 = d$, we have the following subcases: (1.1) For $a + d \geq 0$, we have $s'_3 = s'_1 + b$, i.e., $d' = d$, and so we have the same change in the size at q_3 as at q_1. (1.2) For $a + d < 0$, we have $s'_3 = s_2 + b = s_1 - a \; + b$, and hence $d' = -a$. (2) For $a < 0$, $s_3 = s_2 + b$. In this case, (2.1) for $a + d \geq 0$, $s'_3 = s'_1 + b = s_1 + d + b = s_2 + a \; + d + b$, and so $d' = a \; + d$, and (2.2) for $a + d < 0$, $s'_3 = s_2 + b$, and thus $d' = 0$.

Similarly, when the change is coming from the right, as in Figure 6.6 (b), we have the following cases: (1) For $a \geq 0$, the original size at q_3 is $s_3 = s_1 + b$, and we have the following subcases for the new size: (1.1) For $a - d \geq 0$, $s'_3 = s_1 + b$, and so $d'' = 0$. (1.2) For $a - d < 0$, $s'_3 = s'_2 + b = s_2 + d + b = s_1 - a \; + d + b$, and thus $d'' = -a + d$. (2) For $a < 0$, $s_3 = s_2 + b$. Further, (2.1) for $a - d \geq 0$, $s'_3 = s_1 + b = s_2 + a \; + b$, i.e., $d'' = a$, and (2.2) for $a - d < 0$, $s'_3 = s'_2 + b = s_2 + d + b$, and hence $d'' = d$.

When a change d in the size happens at a child node, at its parent, the change is either eliminated, d' or d'' is 0, stays the same, d' or d'' equals d, becomes $-|a|$ (note that $a \geq 0$ for $d' = -a$, and $a < 0$, for $d'' = a$), or finally, becomes $-|a| + d$. We can now close our construction by showing that the set D of possible changes in the sizes is finite.

Lemma 6.2.4 *For an rTASC A over a set of variables $\mathcal{V}$ and a variable $x \in \mathcal{V}$, the set D of the possible changes in the balance generated by a left tree rotation at x is finite.*

Proof. For D to be infinite, there would have to be a possibility to start with some initial change (either some $-|a|$ or some d_{r_1,r_2}), and then keep modifying it infinitely many times. This can happen only when we use infinitely many times the last case (i.e., $-|a| + d$) from the previous paragraph. Then, we can only start with some d_{r_1,r_2} as for this case to be applied, we need the change in the size at a child node to be positive ($a \geq 0 \;\wedge\; a - d < 0$ for the right case, and $a < 0 \;\wedge\; a + d \geq 0$ for the left case). Note that every time the considered case of propagating the change in the size is applied, we have $d' < d$ or $d'' \leq d$ meaning that the change in the size either does not change or decreases. However, this means that we cannot get an unbounded number of different changes because sooner or later we reach zero and stop generating further changes. □

Note that when we allow the use of two different constants b^1_f and b^2_f in the size function for binary nodes, the resulting class of automata will not be closed with respect to left or right rotations. It may happen that the changes in the balance could diverge, thus we would need an infinite number of compensating constants to be used for the different heights of the possible trees.

Other Operations on Sets of Trees Described by rTASC

Let us now briefly show that in addition to the tree rotations, rTASC are closed with respect to all other operations that we commonly need when dealing with balanced binary trees too. We have listed these operations in Section 6.2.1. We are only giving an informal description of these operations here—their formalisation is, however, straightforward.

Testing and Changing Pointers and Data

We first consider the operation of testing whether two pointer expressions refer to the same node of a tree. Examples of such tests are expressions `x == root` or `x->parent->left == x`. In general, we consider any test of the form e_1==e_2 where e_1, e_2 are of the form v->n_1->n_2->...n_m with $v \in \mathcal{V}$, $m \in \mathbb{N}$, and $n_1, ..., n_m \in \{\texttt{left}, \texttt{right}, \texttt{parent}\}$. Suppose we are given an rTASC A recognising a set S of trees and a pointer equality test c. The rTASC describing the subset S' of S of the trees that meet c is the intersection of A and a TASC A_c encoding c.

To clearly illustrate the construction, let us present an example of A_c for the condition `x->parent->left == x`. We will have rules $f \to q_1$ and $g \to q_2$ for every $f, g \in \Sigma$ such that $\mathtt{x} \in g \setminus f$. We recall that $\Sigma = \mathcal{P}(\mathcal{V} \cup \mathcal{D})$. Then, we have rules $f(q_1, q_1) \to q_1$, $g(q_1, q_1) \to q_2$, $f(q_2, q_1) \to q_3$, $f(q_3, q_1) \to q_3$, and $f(q_1, q_3) \to q_3$ for q_3 being the only accepting state. Here, the pointer referencing pattern gets simply captured in the rule $f(q_2, q_1) \to q_3$.

Second, pointer assignments of the form v' = v->n_1-> n_2->...n_m can be handled by our method, using a simple transformation of the input rTASC which removes v' from the node where it is in the input tree and adds it to the node referenced by v->n_1->n_2->...n_m. Note that we do not treat assignments of the form v->n_1->n_2->...n_m = v'->n'_1->n'_2->...$n'_{m'}$, i.e., destructive updates. We hide these assignments by encoding the effect of the entire procedures in which they appear, i.e., rotations and physical insertion or deletion of nodes. These operations temporarily break the tree shape of the structures being handled by introducing pointer sharing and even cycles. We suppose the correctness of these operations to be checked independently. A generalisation of our method to be able to handle even the internal implementation of these procedures is an interesting subject for further research.

Testing and changing the data contents of the nodes pointed to by some pinter expression of the form v->n_1->n_2->...n_m is an analogy of the pointer reference checking and pointer assignments. However, by changing the data contents of some node (e.g., we recolour some node in a red-black tree), we can change the size of the appropriate subtree. In this case, we have to use the function β from Section 6.2.5 to reflect the change in the balance in the guards of all the rules that can be fired above the node that changed.

Inserting New Nodes

Next, when thinking of the physical insertion of a new leaf node, recall that we suppose the null successors of such memory nodes to be explicitly represented by `null`-labelled nodes in our model. Compared to the real content of the memory, we thus add one layer of nodes. Inserting a new leaf memory node then amounts to replacing one of the null sons of some node by a new, non-null node with two null sons. We abstract here the sortedness property and we just pick randomly the place to insert the new leaf. To encode the operation, we modify the input rTASC by first non-deterministically marking some null node with a pointer variable, i.e., we change its label from $\{\mathtt{null}\}$ to $\{\mathtt{null}, x\}$. Then, we replace all rules $\{\mathtt{null}, x\} \rightarrow q_x$ by rules $\{\mathtt{null}\} \rightarrow q_{\mathtt{null}}$ and $\{d, x\}(q_{\mathtt{null}}, q_{\mathtt{null}}) \xrightarrow{|1| = |2|} q_x$ where d models the initial data content. The addition of the new symbol may change the size of the subtrees above q_x (as, e.g., adding a black node in a red-black tree), and so we have to use the function β from Section 6.2.5 to adjust the guards of the influenced rules.

Deleting Nodes

Finally, the deletion of a frontier node pointed to by some pointer variable y is modelled by removing the rules $\{d, y\}(q, q_{\mathtt{null}}) \xrightarrow{\varphi} q_y$. (Note that a frontier node has at least one null son.) In the remaining rules, we simply replace all the appearances of q_y by all the q states that appeared in the deleted rules. Subsequently, we use again the function β from Section 6.2.5 to handle the changes in the balance resulting from a deletion of a node.

6.2.6 A Case Study: The Red-Black Tree Insertion

To illustrate our methodology, we show how to prove an invariant for the main loop in the procedure RB-Insert. (Note that all the steps are normally to be done *fully automatically.*) This invariant is needed to prove the correctness of the insertion procedure given in Section 6.2.1, that is, given a valid red-black tree as an input to the procedure, the output is also a valid red-black tree. The invariant is the conjunction of the following facts:

1. x is pointing to a non-null node in the tree.
2. If a node is red, then (i) its left son is either black or pointed to by x, and (ii) its right son is either black or pointed to by x. This condition is needed as during the re-balancing of the tree, a red node can temporarily become a son of another red node.
3. The root is either black, or x is pointing to the root.
4. If x is not pointing to the the root and points to a node whose father is red, then x points to a red node.

5. Each maximal path from the root to a leaf contains the same number of black nodes. This is the last condition from the definition of red-black trees from Section 6.2.1.

For presentation purposes, if no guard is specified on a binary rule, we assume it to be $|1| = |2|$. Also, we denote singleton sets by their unique element, e.g., $\{red\}$ by red, and d_x stands for $\{d, x\}$, where $d \in \{red, black, nil\}$. Let $R = \{nil \to q_b, red(q_b, q_b) \to q_r, black(q_{b/r}, q_{b/r}) \to q_b\}$ be a set of transition rules which will be included in each of the automata below. The loop invariant is given by the following rTASC A_1.

$$
\begin{array}{lll}
A_1 : F = \{q_{rx}, q_{bx}, q'_{bx}\}, & & \\
\Delta = R \cup \{ & black_x(q_{b/r}, q_{b/r}) \to q_{bx}\ (1), & black(q_{bx/rx}, q_{b/r}) \to q'_{bx}\ (2), \\
& black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}\ (3), \\
& black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}, & red_x(q_b, q_b) \to q_{rx}, \\
& red(q'_{bx}, q_b) \to q'_{rx}, & red(q_b, q'_{bx}) \to q'_{rx}, \\
& red(q_{rx}, q_b) \to q'_{rx}\ (4), & red(q_b, q_{rx}) \to q'_{rx}\ (5)\}
\end{array}
$$

Intuitively, q_b labels black nodes and q_r red nodes which do not have a node pointed to by x below them. q_{bx} and q_{rx} mean the same except that they label a node which is pointed to by x. Primed versions of q_{bx} and q_{rx} are used for nodes which have a subnode pointed to by x. In the following, this intuitive meaning of states will be changed by the program steps. We refer to the pseudo-code of Section 6.2.1.

If the loop entrance condition `x!= root && x->parent->colour == red` is true, we obtain a new automaton A_2. It is given by modifying A_1 as follows: $F = \{q'_{bx}\}$ and the rules (1), (2) and (3) are removed.

If the condition `x->parent == x->parent->parent->left` is true, we take A_2, change rule (4) to $red(q_{rx}, q_b) \to q''_{rx}$, rule (5) to $red(q_b, q_{rx}) \to q''_{rx}$ and add a rule $black(q''_{rx}, q_{b/r}) \to q'_{bx}$ (6) to obtain A_3. Now, q''_{rx} accepts the father of the node pointed by `x` and q'_{rx} its grandfather.

If the condition `x->parent->parent->right->colour == red` holds, we obtain the automaton A_4 that is like A_3 except for rule (6) changed into $black(q''_{rx}, q_r) \to q'_{bx}$.

The recolouring step `x->parent->colour = black` changes some guards on rules and leads to a propagation of the change through the automaton. The result is A_5:

$$
\begin{array}{ll}
A_5 :\ F = \{q'_{bx}\}, \Delta = R\ \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \xrightarrow{|1| = |2| + 1} q'_{bx}, & red_x(q_b, q_b) \to q_{rx}, \\
\ black(q_{b/r}, q'_{bx/rx}) \xrightarrow{|1| + 1 = |2|} q'_{bx}, & red(q'_{bx}, q_b) \xrightarrow{|1| = |2| + 1} q'_{rx}, \\
\ black(q''_{rx}, q_r) \xrightarrow{|1| = |2| + 1} q'_{bx}\ (7), & red(q_b, q'_{bx}) \xrightarrow{|1| + 1 = |2|} q'_{rx}, \\
\ black(q_{rx}, q_b) \to q''_{rx}, & black(q_b, q_{rx}) \to q''_{rx}\}
\end{array}
$$

After the recolouring step `x->parent->parent->right->colour = black`, we get A_6, which is A_5 where we change rule (7) to $black(q''_{rx}, q_b) \rightarrow q'_{bx}$. Note that no propagation is needed in this case.

After the recolouring step `x->parent->parent->colour = red` that introduces changes on guards, and the propagation of these changes, we obtain:

$$\begin{array}{lll} A_7: \ F = \{q'_{bx}\}, \Delta = R \ \cup & & \\ \{black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, & black(q_{rx}, q_b) \rightarrow q''_{rx}, \\ \ black(q_b, q_{rx}) \rightarrow q''_{rx}, & red_x(q_b, q_b) \rightarrow q_{rx} \ (8), & red(q'_{bx}, q_b) \rightarrow q'_{rx}, \\ \ red(q''_{rx}, q_r) \rightarrow q'_{bx} \ (9), & red(q_b, q'_{bx}) \rightarrow q'_{rx}\} & \end{array}$$

After `x = x->parent->parent`, we get A_8 derived from A_7 by changing rule (8) to $red(q_b, q_b) \rightarrow q_{rx}$ and rule (9) to $red_x(q''_{rx}, q_b) \rightarrow q'_{bx}$.

This takes care of Case 1, and one can then check that $\mathcal{L}(A_8) \subseteq \mathcal{L}(A_1)$.

For Case 2, we have to go back to automaton A_3 and apply the fact that the conditional `x->parent->parent->right->colour == red` is false, i.e., in other words, the condition `x->parent->parent->right->colour == black` must be true. The result is:

$$\begin{array}{lll} A_9: \ F = \{q'_{bx}\}, & & \\ \Delta = R \ \cup \{ & black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, \\ & black(q''_{rx}, q_b) \rightarrow q'_{bx}, & red_x(q_b, q_b) \rightarrow q_{rx} \ (11), \\ & red(q'_{bx}, q_b) \rightarrow q'_{rx}, & red(q_b, q'_{bx}) \rightarrow q'_{rx}, \\ & red(q_b, q_{rx}) \rightarrow q''_{rx} \ (12), & red(q_{rx}, q_b) \rightarrow q''_{rx} \ (10)\} \end{array}$$

After the condition `x == x->parent->right`, A_9 is changed into A_{10} by removing rule (10). After `x = x->parent`, A_{10} is changed into A_{11} by changing rule (11) to $red(q_b, q_b) \rightarrow q_{rx}$ and rule (12) to $red_x(q_b, q_{rx}) \rightarrow q''_{rx}$.

Now, the operation `Left-Rotate(T,x)` introduces new states and transitions, and we get the TASC A_{12}. Notice that no rebalancing is necessary.

$$\begin{array}{lll} A_{12}: \ F = \{q'_{bx}\}, & & \\ \Delta = R \ \cup \{ & black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, \\ & black(q_{rot2}, q_b) \rightarrow q'_{bx}, & red_x(q_b, q_b) \rightarrow q_{rot1}, \\ & red(q'_{bx}, q_b) \rightarrow q'_{rx}, & red(q_b, q'_{bx}) \rightarrow q'_{rx}, \\ & red(q_{rot1}, q_b) \rightarrow q_{rot2}\} & \end{array}$$

After `x->parent->colour = black` and a propagation of the changes in the balance, we obtain:

$$\begin{array}{lll} A_{13}: \ F = \{q'_{bx}\}, & & \\ \Delta = R \ \cup \ \{ & black(q'_{bx/rx}, q_{b/r}) \xrightarrow{|1| = |2| + 1} q'_{bx}, & red_x(q_b, q_b) \rightarrow q_{rot1}, \\ & black(q_{b/r}, q'_{bx/rx}) \xrightarrow{|1| + 1 = |2|} q'_{bx}, & red(q'_{bx}, q_b) \xrightarrow{|1| = |2| + 1} q'_{rx}, \\ & black(q_{rot2}, q_b) \xrightarrow{|1| = |2| + 1} q'_{bx}, & red(q_b, q'_{bx}) \xrightarrow{|1| + 1 = |2|} q'_{rx}, \\ & black(q_{rot1}, q_b) \rightarrow q_{rot2}\} & \end{array}$$

After `x->parent->parent->colour = red`, we obtain:

$$\begin{array}{lll} A_{14}: \ F = \{q'_{bx}\}, & & \\ \Delta = R \ \cup \ \{ & black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, & red_x(q_b, q_b) \rightarrow q_{rot1}, \\ & black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, & red(q'_{bx}, q_b) \rightarrow q'_{rx}, \\ & red(q_{rot2}, q_b) \xrightarrow{|1| = |2| + 1} q'_{bx}, & red(q_b, q'_{bx}) \rightarrow q'_{rx}, \\ & black(q_{rot1}, q_b) \rightarrow q_{rot2}\} & \end{array}$$

Finally, after `Right-Rotate(T,x->parent->parent)`, we get:

$$\begin{array}{lll} A_{15}: \ F = \{q'_{bx}\}, & & \\ \Delta = R \ \cup \ \{ & black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx} \\ & black(q_{b/r}, q_{rot4}) \rightarrow q'_{bx}, & black(q_{rot4}, q_{b/r}) \rightarrow q'_{bx}, \\ & black(q_{rot1}, q_{rot3}) \rightarrow q_{rot4}, & red_x(q_b, q_b) \rightarrow q_{rot1}, \\ & red(q'_{bx}, q_b) \rightarrow q'_{rx}, & red(q_b, q'_{bx}) \rightarrow q'_{rx}, \\ & red(q_{rot4}, q_b) \rightarrow q'_{rx}, & red(q_b, q_b) \rightarrow q_{rot3}, \\ & red(q_b, q_{rot4}) \rightarrow q'_{rx}\} & \end{array}$$

Then, it can be checked that $\mathcal{L}(A_{15}) \subseteq \mathcal{L}(A_1)$. Case 3 of the insertion procedure is very similar to Case 2, and so we omit it here.

6.3 Summary and Further Work

In the chapter, we have first briefly discussed multiple interesting approaches to symbolic formal verification based on using various kinds of more-than-regular languages. Then, we have discussed in detail a work that is our own contribution in this area obtained in a close collaboration with our foreign partners.

In particular, our contribution consists in a proposal of a method for semi-algorithmic verification of programs that manipulate balanced trees. The approach is based on specifying program preconditions, postconditions, and invariants as sets of trees recognised by a novel class of extended tree automata called TASC. TASC come with interesting closure properties and a decidable emptiness problem. Moreover, the semantics of tree-updating programs can be effectively represented as modifications on the internal structures of TASC. The framework has been validated on a case study consisting of the node insertion procedure for red-black trees. Precisely, we verified that given a balanced red-black tree on the input to the insertion procedure, the output is again a balanced red-black tree.

In the future, we plan to implement the method to be able to perform more case studies. An interesting subject for further research is then extending the method to a fully automatic one. For this, a suitable acceleration method for the reachability computation on TASC is needed. Also, it is interesting to try

to generalise the method to handle even the internals of low-level manipulations that temporarily break the tree shape of the considered structures (e.g., by lifting the technique to work over tree automata extended with routing expressions describing additional pointers over the tree backbone as considered in [BHRV06b]).

Moreover, apart from the above mentioned possible future work on TASC, there is of course a large space for proposing and experimenting with various new automata and languages (in general not limiting their application to dealing with balanced dynamic linked data structures). There even already exist some classes of languages and automata (some proposed relatively recently) that appear quite promising for automated symbolic verification extending the concept of regular model checking (or more specifically, abstract regular model checking). Among such classes, we can list, e.g., visibly push-down languages [AM04], AC-tree languages [Ohs01], or languages of various kinds of automata with restricted counters. Some of these classes have already been used for symbolic verification (as mentioned in Section 6.1), but not in a way being an extension of the generic regular model checking framework. The same then applies for combining the use of these automata with automated abstraction with counterexample-guided refinement schemas. The use of abstraction in this context might at the same time be quite advantageous as operations on such automata are often much more expensive than on classical finite-state automata.

7

Conclusion

We have included a concrete summary and a discussion of possible future work into every chapter of the previous text. Let us, however, briefly summarise the contents of the work once again and let us also sum up and generalise the possible directions for future research.

7.1 Summary

We have concentrated on two approaches to applying model checking in formal verification of infinite-state systems. Namely, we have considered the use of *cut-offs* and the use of *regular model checking* and some of its extensions. We have payed a lot of attention to the area of verification of programs manipulating dynamic linked data structures as one of the areas where regular model checking can be quite successfully applied. We have presented in detail some of our work contributing to these areas, but we have also tried to give a careful overview of other existing approaches.

As a part of our contribution, we have presented multiple cut-off results for verification of parameterised networks of processes. In particular, we considered networks of processes with shared resources available through a possibly prioritised FIFO resource granting discipline, which is often used in practice. The cut-offs covered linear-time safety (including mutual exclusion), liveness (and in particular absence of starvation) as well as deadlockability properties that are the most frequent properties of interest in the given setting.

Then, we have discussed our proposal of abstract regular model checking combining regular model checking with the CEGAR approach and allowing regular model checking to be often performed much more efficiently than using other approaches proposed in the literature. The method has also been generalised to abstract regular tree model checking with similar results. We have further proposed a method of regular model checking based on inference of regular languages from their samples, which guarantees termination for all the cases when the studied system has a regular state space.

Next, we have described a way how (abstract) regular model checking can be applied to verification of programs manipulating dynamic singly-linked data structures leading to a fully automated and quite efficient way of verifying many of their properties. Moreover, this approach has recently been successfully generalised by using abstract regular tree model checking to handle even much more general structures, some of which were not previously handled by a method of a similar degree of automation.

Finally, we have also studied the use of non-regular languages for symbolic automata-based verification leading to a proposal of a novel class of tree automata with size constraints with an application to verification of programs manipulating balanced tree structures.

7.2 Future Research Directions

Optimising the Existing Techniques

When we try to summarise the possible future research directions mentioned in the particular preceding chapters, we obtain that one of the possible general future research directions is trying to *optimise the existing techniques* in various ways.

In the domain of cut-offs, the optimisation implies trying to reduce their size in the cases where they are known not to be optimal. In the case of automata-based symbolic model checking, one can think of new abstractions and also about more efficient implementations of the underlying automata libraries and the basic language-theoretic operations on automata. This implies, e.g., a careful use of the latest BDD technology, heuristics like guides in guided tree automata in Mona [BKR97], and even looking for novel ways how to deal with the basic automata operations (checking for emptiness, inclusion, etc.). An interesting idea is, for instance, trying to deal with non-deterministic automata instead of deterministic ones, which could allow us to bypass the expensive determinisation step as proposed recently in [WDHR06].

Of course, optimising the existing techniques is not a problem of cut-offs and automata-based symbolic model checking only, a similar need can be identified for the other works targeting automated verification of infinite-state systems too. Many various ways have been proposed for coping with the state explosion problem in finite-state model checking, and it is interesting whether similar advances and heuristics can be provided for the more recent infinite-state approaches too. Here, as a quite promising research direction, we can mention, e.g., combining the infinite-state approaches with principles of modular verification. Moreover, an interesting subject is also the area of combining infinite-state approaches with efficient finite-state ones applied on the finite-state part of states of the examined systems.

More Specialised Techniques

As a particular direction for optimising the efficiency of the existing techniques, one can consider development of *more specialised techniques* exploiting the richer knowledge on the domain being solved for attaining a better performance. The specialisation may consider narrower classes of both systems to be verified as well as their properties. The specialisation can be exploited on different levels starting from the applied abstractions and symbolic representations down to the underlying low-level representation and operations such as automata and BDDs. The abstraction can be tuned to deal, e.g., with a certain shape of dynamic linked data structures common in practice while detecting when this shape is not preserved (as considered for some cases, e.g., in [BHMV05, LAIS06]) and giving up in such a situation. As another example we can note that although the worst-case complexity of the basic operations on automata operations is, of course, known for a long time, one could try to think of categorising automata according to various criteria, identifying which classes of automata exist in which domains and trying to optimise the operations for these classes—such a direction is considered, e.g., in [TV05].

Dealing with More General Systems and/or Properties

Another interesting research direction is in general trying to handle *more complex structures and/or properties* than so-far. This includes dealing with systems with more different sources of infinity as, e.g., various combinations of recursion, unbounded counters, unbounded concurrency, and/or dynamic data structures. A difficult problem is also handling systems whose states have a complex graph structure like, for instance, various complex dynamic data structures such as threaded AVL trees (i.e., AVL trees with nodes linked additionally in the post-order way).

For handling such complex systems, one has to think of appropriate symbolic state space representation structures (either new ones or combinations and/or extensions of the existing ones) and of suitable algorithms for handling such representations. In this context, developing and applying abstractions in a similar way as, e.g., in abstract regular model checking may become even more crucial than for the currently used representations as with the rising complexity of the representations, the price of operations on them will quite likely rise as well.

Next, apart from handling basic safety properties, it is always a challenge to handle liveness/termination properties and properties with various quantitative features (balancedness, etc.). In abstract symbolic approaches applied for liveness/termination checking, it is then, e.g., a challenge how to detect and remove spurious infinite (looping) counterexamples which is significantly more difficult than for finite spurious counterexamples in the case of safety checking.

Model Extraction

Finally, apart from studying various symbolic state space representation structures and algorithms for dealing with them, another challenge is sometimes also how to automatically build models of systems suitable for verification via such formalisms from the source description of the systems being checked. Such models can, e.g., have the form of automata extended with counters, queues, arrays, etc. Techniques like predicate abstraction are then to be generalised to produce not only finite-state models but such extended models.

References

[AAB99] P.A. Abdulla, A. Annichini, and A. Bouajjani. Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol. In *Proc. of TACAS'99*, volume 1579 of *LNCS*. Springer, 1999.

[ABB01] P.A. Abdulla, L. Boasson, and A. Bouajjani. Effective Lossy Queue Languages. In *Proc. of ICALP'01*, volume 2076 of *LNCS*. Springer, 2001.

[ABH+97] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[ABJ98] P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[ABJN99] P. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parametrized System Verification. In *Proc. of CAV'99*, volume 1633 of *LNCS*. Springer, 1999.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-Checking in Dense Real-Time. *Information and Computation*, 104:2–34, 1993. First appeared in Proc. of LICS'90.

[AD94] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994. First appeared in Proc. of ICALP'90.

[AdJN02] P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Regular Model Checking Made Simple and Efficient. In *Proc. of CONCUR'02*, volume 2421 of *LNCS*. Springer, 2002.

[AdJN03] P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Algorithmic Improvements in Regular Model Checking. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.

[AED02] T. Arts, C.B. Earle, and J. Derrick. Verifying Erlang Code: A Resource Locker Case-Study. In *Proc. of FME'02*, volume 2391 of *LNCS*. Springer, 2002.

[AJ96a] P. Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. *Information and Computation*, 127(2):91–101, 1996.

[AJ96b] P.A. Abdulla and B. Jonsson. Undecidable Verication Problems for Programs with Unreliable Channels. *Information and Computation*, 130(1):71–90, 1996.

[AJMd02] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[AJN+04] P.A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular Model Checking for MSO + LTL. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[AK86] K. Apt and D. Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Information Processing Letters*, 22(6):307–309, 1986.

[ALdA05] P.A. Abdulla, A. Legay, J. d'Orso, and A.Rezine. Simulation-Based Iteration of Tree Transducers. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[AM04] R. Alur and P. Madhusudan. Visibly Pushdown Languages. In *Proc. of STOC'04*. ACM Press, 2004.

[AMN05] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[Ang87] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L.D. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *Proc. of CAV'01*, volume 2102 of *LNCS*. Springer, 2001.

[BAN04] S. Bardin, A.Finkel, and D. Nowak. Toward Symbolic Verification of Programs Handling Pointers. In *Proc. of AVIS'04*, 2004.

[BB04] C. Bartzis and T. Bultan. Widening Arithmetic Automata. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[BBH+06a] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[BBH+06b] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. Technical Report TR-2006-3, Verimag, UJF/CNRS/INPG, Grenoble, France, 2006.

[BBLS00] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Prameterized Networks. In *Proc. of TACAS 2000*, volume 1785 of *LNCS*. Springer, 2000.

[BCE+05] P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proc. of COSMICAH'05*, Technical report RR-05-04. Queen Mary, University of London, 2005.

[BCK01] P. Baldan, A. Corradini, and B. König. A Static Analysis Technique for Graph Transformation Systems. In *Proc. of CONCUR'01*, volume 2154 of *LNCS*. Springer, 2001.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proc. of CONCUR'97*, LNCS. Springer, 1997.

[BESS05] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejček. Reachability Analysis of Multithreaded Software with Asynchronous Communication. In *Proc. of FSTTCS'05*, volume 3821 of *LNCS*. Springer, 2005.

[BF04] S. Bardin and A. Finkel. Composition of Accelerations to Verify Infinite Heterogeneous Systems. In *Proc. of ATVA'04*, volume 3299 of *LNCS*. Springer, 2004.

[BFL06] S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proc. of AVIS'06*, 2006.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The Power of QDDs. In *Proc. of SAS'97*, volume 1302 of *LNCS*. Springer, 1997.

[BH99] A. Bouajjani and P. Habermehl. Symbolic Reachability Analysis of FIFO-Channel Systems with Nonregular Sets of Configurations. *Journal of Theoretical Computer Science*, 221(1/2):221–250, 1999.

[BHM03] A. Bouajjani, P. Habermehl, and R. Mayr. Automatic Verification of Recursive Procedures with One Integer Parameter. *Theoretical Computer Science*, 1–3:85–106, 2003. A preliminary version was presented at MFCS'01.

[BHMV05] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[BHRV05] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. In *Proc. of Infinity'05*, 2005.

[BHRV06a] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.

[BHRV06b] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[BHT05] D. Beyer, T.A. Henzinger, and G. Théoduloz. Lazy Shape Analysis. Technical Report HMTC-REPORT-2005-006, EPFL, Lausanne, Switzerland, 2005.

[BHV02] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parameterized Concurrent Systems with Resource Sharing. A research report of the ADVANCE project (IST FET project No. IST-1999-29082). LIAFA, University Paris 7, France, 2002.

[BHV03] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parametric Concurrent Systems with Prioritized FIFO Resource Management. In *Proc. of Concur'03*, volume 2761 of *LNCS*. Springer, 2003.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[BHV06] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parametric Concurrent Systems with Prioritized FIFO Resource Management. *Formal Methods in Systems Design*, 2006. Submitted for a review.

[BI05] M. Bozga and R. Iosif. Quantitative Verification of Programs with Lists. In *Proc. of VISSAS'05*, 2005.

[BIL03] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[BJW01] B. Boigelot, S. Jodogne, and P. Wolper. On the Use of Weak Automata for Deciding Linear Arithmetic with Integer and Real Variables. In *Proc. of IJCAR'01*, volume 2083 of *LNCS*. Springer, 2001.

[BKMW01] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[BKR97] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for Guided Tree Automata. In *Proc. of WIA'96*, volume 1260 of *LNCS*. Springer, 1997.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[BLW03] B. Boigelot, A. Legay, and P. Wolper. Iterating Transducers in the Large. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.

[BLW04] B. Boigelot, A. Legay, and P. Wolper. Omega-Regular Model Checking. In *Proc. of TACAS'04*, volume 2988 of *LNCS*. Springer, 2004.

[BLW05] A. Bouajjani, A. Legay, and P. Wolper. Handling Liveness Properties in (ω-)Regular Model Checking. *ENTCS*, 138:101–115, 2005. A preliminary version was presented at Infinity'04.

[BMOT05] A. Bouajjani, M. Mueller-Olm, and T. Touili. Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems. In *Proc. of Concur'05*, volume 3653 of *LNCS*. Springer, 2005.

[BP94] B. Boigelot and P. Wolper. Symbolic Verification with Periodic Sets. In *Proc. of CAV'94*, volume 818 of *LNCS*. Springer, 1994.

[BP96] B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[BPR01] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001.

[BPZ05] I. Balaban, A. Pnueli, and L.D. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.

[BR01] T. Ball and S.K. Rajamani. The SLAM Toolkit. In *Proc. of CAV'01*, volume 2102 of *LNCS*. Springer, 2001.

[BRW98] B. Boigelot, S. Rassart, and P. Wolper. On the Expressiveness of Real and Integer Arithmetic Automata. In *Proc. of ICALP'98*, volume 1443 of *LNCS*. Springer, 1998.

[Bry86] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BS97] O. Burkart and B. Steffen. Model Checking the Full Modal mu-Calculus for Infinite Sequential Processes. In *Proc. of ICALP'97*, volume 1256 of *LNCS*. Springer, 1997.

[BT02] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[BT03] A. Bouajjani and T. Touili. Reachability Analysis of Process Rewrite Systems. In *Proc. of FSTTCS'03*, volume 2914 of *LNCS*. Springer, 2003.

[BT05] A. Bouajjani and T. Touili. On Computing Reachability Sets of Process Rewrite Systems. In *Proc. of RTA'05*, volume 3467 of *LNCS*. Springer, 2005.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*. ACM Press, 1977.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. of POPL'79*. ACM Press, 1979.

[CC05] H. Comon and V. Cortier. Tree Automata with One Memory, Set Constraints and Cryptographic Protocols. *Theoretical Computer Science*, 331, 2005.

[CCG+04] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design*, 25(2–3):129–166, 2004.

[CCST05] S. Chaki, E. Clarke, N. Sinha, and P. Thati. Automated Assume-Guarantee Reasoning for Simulation Conformance. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[CDG+05] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005.
URL: `http://www.grappa.univ-lille3.fr/tata`.

[CDOY06] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*. Springer, 1981.

[ČEV05] M. Češka, P. Erlebach, and T. Vojnar. Pattern-Based Verification of Programs with Extended Linear Linked Data Structures. In *Proc. of AVOCS'05*, 2005.

[ČEV06] M. Češka, P. Erlebach, and T. Vojnar. Pattern-Based Verification of Programs with Extended Linear Linked Data Structures. *ENTCS*, 145:113–130, 2006. A preliminary version was presented at AVOCS'05.

[ČEV07] M. Češka, P. Erlebach, and T. Vojnar. Pattern-based Verification for Trees. In *Proc. of EUROCAST'07*, volume 4739 of *LNCS*. Springer, 2007.

[CFI96a] G. Cécé, A. Finkel, and S. Purushothaman Iyer. Unreliable Channels are Easier to Verify than Perfect Channels. *Information and Computation*, 124(1):20–31, 1996.

[CFI96b] G. Cécé, A. Finkel, and S.P. Iyer. Unreliable Channels Are Easier to Verify Than Perfect Channels. *Information and Computation*, 141(1), 1996.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. In *Proc. of CAV'93*, volume 697 of *LNCS*. Springer, 1993.

[CGJ+00a] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[CGJ+00b] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CLM89] E.M. Clarke, D. Long, and K.L. McMillan. Compositional Model Checking. In *Proc. of LICS'89*. IEEE Press, 1989.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[Col03] T. Colcombet. Rewriting in the Partial Algebra of Typed Terms Modulo AC. *ENTCS*, 68:1–15, 2003.

[Cou81] P. Cousot. Semantic Foundations of Program Analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[CPR05] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*, 2005.

[CR00] S.J. Creese and A.W. Roscoe. Data Independent Induction over Structured Networks. In *Proc. of PDPTA 2000*. CSREA Press, 2000.

[CTTV04] E.M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by Network Decomposition. In *Proc. of CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.

[DBCO06] D. Distefano, Josh Berdine, Byron Cook, and P.W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[DBIK04] Z. Dang, T. Bultan, O.H. Ibarra, and R.A. Kemmerer. Past Pushdown Timed Automata and Safety Verification. *Theoretical Computer Science*, 313:57–71, 2004.

[DD02] S. Das and D.L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Proc. of FMCAD'02*, 2002.

[DEG06] J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[Deu92] A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, University Paris VI, Paris, France, 1992.

[Deu94] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *Proc. of PLDI'94*. ACM Press, 1994.

[Dil89] D.L. Dill. Timing Assumptions and Verification of Finite-state Concurrent Systems. In *Proc. of Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer, 1989.

[DL02] S. Dal Zilio and D. Lugiez. Multitree Automata, Presburger's Constraints and Tree Logics. Technical Report 08-2002, LIF, 2002.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating Transducers. In *Proc. of CAV'01*, volume 2102 of *LNCS*. Springer, 2001.

[DN03] D. Dams and K.S. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*. Springer, 2003.

[DOY06] D. Distefano, P.W. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[Dup96] P. Dupont. Incremental Regular Inference. In *Grammatical Inference: Learning Syntax from Sentences*, volume 1147 of *LNAI*, 1996.

[EH86] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'Not Never' Revisited: On Branching Versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[EK00] E.A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. In *Proc. of CADE 2000*, volume 1831 of *LNCS*. Springer, 2000.

[EK02] E.A. Emerson and V. Kahlon. Model Checking Large-Scale and Parameterized Resource Allocation Systems. In *Proc. of TACAS'02*, volume 2280 of *LNCS*. Springer, 2002.

[EK03] E.A. Emerson and V. Kahlon. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In *Proc. of CHARME'03*, volume 2860 of *LNCS*. Springer, 2003.

[EK04] E.A. Emerson and V. Kahlon. Parameterized Model Checking of Ring-based Message Passing Systems. In *Proc. of CS'04*, volume 3210 of *LNCS*. Springer, 2004.

[EM04] D. Engler and M. Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *Proc. of VMCAI'04*, volume 2937 of *LNCS*. Springer, 2004.

[EMS00] J. Elgaard, A. Møller, and M.I. Schwartzbach. Compile Time Debugging of C Programs Working on Trees. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer, 2000.

[EN95] E.A. Emerson and K.S. Namjoshi. Reasoning about Rings. In *Proc. of POPL'95*. ACM Press, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[Eng75] J. Engelfriet. Bottom-up and Top-down Tree Transformations—A Comparison. *Mathematical System Theory*, 9:198–231, 1975.

[ES97] E.A. Emerson and A.P. Sistla. Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997.

[Esp02] J. Esparza. Grammars as Processes. In *Formal and Natural Computing*, volume 2300 of *LNCS*. Springer, 2002.

[EV05] P. Erlebach and T. Vojnar. Automated Formal Verification of Programs with Dynamic Data Structures Using State-of-the-Art Tools. In *Proc. of ISIM'05*. MARQ Ostrava, 2005.

[EVW97] K. Etessami, M.Y. Vardi, and T. Wilke. First-Order Logic with Two Variables and Unary Temporal Logic. In *Proc. of LICS'97*. IEEE Computer Society Press, 1997.

[Fin94] A. Finkel. Decidability of the Termination Problem for Completely Specified Protocols. *Distributed Computing*, 7(3):129–135, 1994.

[FL02] A. Finkel and J. Leroux. How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In *Proc. of FSTTCS'02*, volume 2556 of *LNCS*. Springer, 2002.

[FO97] L. Fribourg and H. Olsen. Reachability Sets of Parametrized Rings as Regular Languages. *ENTCS*, 9, 1997. A preliminary version was presented at Infinity'97.

[FP01] D. Fisman and A. Pnueli. Beyond Regular Model Checking. In *Proc. of FSTTCS'01*, volume 2245 of *LNCS*. Springer, 2001.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. *ENTCS*, 9, 1997. A preliminary version was presented at Infinity'97.

[Gei91] D. Geidmanis. Unsolvability of the Emptiness Problem for Alternating 1-way Multi-head and Multi-tape Finite Automata over Single-letter Alphabet. In *Computers and Artificial Intelligence*, volume 10, 1991.

[Gen] T. Genet. Timbuk: A Tree Automata Library. URL: `http://www.irisa.fr/lande/genet/timbuk`.

[GK00] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proc. of CADE'00*, volume 1831 of *LNCS*. Springer, 2000.

[GLP06] O. Grinchtein, M. Leucker, and N. Piterman. Inferring Network Invariants Automatically. In *Proc. of IJCAR'06*, volume 4130 of *LNCS*. Springer, 2006.

[God91] P. Godefroid. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proc. of CAV'91*, volume 575 of *LNCS*. Springer, 1991.

[GS92] S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[GT01] T. Genet and V. Viet Triem Tong. Reachability Analysis of Term Rewriting Systems with Timbuk. In *Proc. of LPAR'01*, volume 2250 of *LNAI*. Springer, 2001.

[HIRV07] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. In *Proc. of ATVA'07*, volume 4762 of *LNCS*. Springer, 2007.

[HIV05] P. Habermehl, R. Iosif, and T. Vojnar. Automata-Based Verification of Programs with Tree Updates. Technical Report TR-2005-16, Verimag, UJF/CNRS/INPG, Grenoble, France, 2005.

[HIV06] P. Habermehl, R. Iosif, and T. Vojnar. Automata-Based Verification of Programs with Tree Updates. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. of POPL'02*. ACM Press, 2002.

[HJMS03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *Proc. of 10th SPIN Workshop*, volume 2648 of *LNCS*. Springer, 2003.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Information and Computation*, 111:193–244, 1994. First appeared in Proc. of LICS'92.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[HV04] P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. In *Proc. of Infinity'04*, 2004.

[HV05] P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. *ENTCS*, 138:21–36, 2005. A preliminary version was presented at Infinity'04.

[Iba78] O.H. Ibarra. Reversal-bounded Multicounter Machines and Their Decision Problems. *Journal of the ACM*, 25:116–133, 1978.

[Iba00] O.H. Ibarra. Reachability and Safety in Queue Systems. In *Proc. of CIAA'00*, volume 2088 of *LNCS*. Springer, 2000.

[ID96a] C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Journal of Formal Methods in System Design*, 9(1/2):41–76, 1996.

[ID96b] C.N. Ip and D.L. Dill. Verifying Systems with Replicated Components in Murφ. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[IDP03] O.H. Ibarra, Z. Dang, and P. San Pietro. Verification in Loosely Synchronous Queue-connected Discrete Timed Automata. *Theoretical Computer Science*, 290:1713–1735, 2003.

[ISD+02] O.H. Ibarra, J. Su, Z. Dang, T. Bultan, and R.A. Kemmerer. Counter Machines and Verification Problems. *Theoretical Computer Science*, 289:165–189, 2002.

[JJKS97] J.L. Jensen, M.E. Jørgensen, N. Klarlund, and M.I. Schwartzbach. Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In *Proc. of PLDI'97*, 1997.

[JN00] B. Jonsson and M. Nilsson. Transitive Closures of Regular Relations for Verifying Infinite-State Systems. In *Proc. of TACAS'00*, volume 1785 of *LNCS*. Springer, 2000.

[Jon81] H.B.M. Jonkers. Abstract Storage Structures. In *Algorithmic Languages*. IFIP, 1981.

[KIG05] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about Threads Communicating via Locks. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[KK06] B. König and V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[Kle87] S. Kleene. *Introduction to Mathematics*. North-Holland, 1987.

[KM95] R.P. Kurshan and K.L. McMillan. A Structural Induction Theorem for Processes. *Information and Computation*, 117(1), 1995.

[KM01] N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.

[KMe00a] M. Kaufmann, P. Manolios, and J. Strother Moore (eds.). *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[KMe00b] M. Kaufmann, P. Manolios, and J. Strother Moore (eds.). *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. *Theoretical Computer Science*, 256(1–2), 2001.

[Koz83] D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP88] S. Katz and D. Peled. An Efficient Verification Method for Parallel and Distributed Programs. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Workshop*, volume 354 of *LNCS*. Springer, 1988.

[KS93] N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.

[KvS04] M. Křetínský, V. Řehák, and J. Strejček. Extended Process Rewrite Systems: Expressiveness and Reachability. In *Proc. of Concur'04*, volume 3170 of *LNCS*. Springer, 2004.

[LAIS06] T. Lev-Ami, N. Immerman, and M. Sagiv. Fast and Precise Abstraction for Shape Analysis. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[Lan92] K. J. Lang. Random DFA's can be Approximately Learned from Sparse Uniform Examples. In *Proc. of the 5th ACM Workshop on Computational Learning Theory*. ACM Press, 1992.

[LAS] The Liège Automata-based Symbolic Handler (LASH). URL: `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[Ler06] J. Leroux. Regular Acceleration for Number Decision Diagrams. Technical Report 1385-06, LABRI, Bordeaux, France, 2006.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. In *Proc. of POPL'97*. ACM Press, 1997.

[LQ06] S.K. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Proc. of POPL'06*. ACM Press, 2006.

[LRS05] A. Loginov, T. Reps, and M. Sagiv. Abstraction Refinement via Inductive Learning. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[Lug05] D. Lugiez. Multitree Automata That Count. *Theoretical Computer Science*, 333:225–263, 2005.

[LYY05] O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proc. of ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.

[May00a] R. Mayr. Process Rewrite Systems. *Information and Computation*, 156(1):264–286, 2000.

[May00b] Richard Mayr. Undecidable Problems in Unreliable Computations. In *Latin American Theoretical Informatics*, pages 377–386, 2000.

[McM05] K.L. McMillan. Applications of Craig Interpolants in Model Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[Mon03] D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. *Science of Computer Programming*, 47(2–3), 2003.

[MQS00] K.L. McMillan, S. Qadeer, and J.B. Saxe. Induction in Compositional Model Checking. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[MS01] A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. Also in SIGPLAN Notices 36(5), 2001.

[MS02] B. Masson and P. Schnoebelen. On Verifying Fair Lossy Channel Systems. In *Proc. of MFCS'02*, volume 2420 of *LNCS*. Springer, 2002.

[MSS86] D.E. Mueller, A. Saoudi, and P.E. Schupp. Alternating Automata, the Weak Monadic Theory of the Tree and its Complexity. In *Proc. of ICALP'86*, volume 226 of *LNCS*. Springer, 1986.

[MSZ07] Z. Manna, H. B. Sipma, and T. Zhang. Verifying Balanced Trees. In *Proc. of LFCS'07*, volume 4514 of *LNCS*. Springer, 2007.

[MYRS05] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.

[NDQC07] H. H. Nguyen, C. David, S. Qin, and W. N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *Proc. of VMCAI'07*, volume 4349 of *LNCS*. Springer, 2007.

[Nil00] M. Nilsson. Regular Model Checking. Licentiate Thesis, Uppsala University, Sweden, 2000.

[Nil05] M. Nilsson. *Regular Model Checking*. PhD thesis, Uppsala University, 2005.

[NPW05] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2005.

[OG92] J. Oncina and P. Garcia. Inferring Regular Languages in Polynomial Update Time. In *Pattern Recognition and Image Analysis*, 1992.

[Ohs01] H. Ohsaki. Beyond Regularity: Equational Tree Automata for Associative and Commutative Theories. In *Proc. of CSL'01*, volume 2142 of *LNCS*. Springer, 2001.

[OSRSC01] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, California, USA, 2001. URL: `http://pvs.csl.sri.com/`.

[OT05] H. Ohsaki and T .Takai. ACTAS: A System Design for Associative and Commutative Tree Automata Theory. *ENTCS*, 124:97–111, 2005.

[Pac87] J.K. Pachl. Protocol Description and Analysis based on a State Transition Model with Channel Expressions. In *Protocol Verification, Specification, and Testing VII*, 1987.

[Par05] S. Parduhn. Algorithm Animation Using Shape Analysis with Special Regard to Binary Trees. Master's thesis, Universität des Saarlandes, Fachrichtung 6.2 – Informatik, 2005.

[Pet95] H. Petersen. Alternation in Simple Devices. In *Proc. of ICALP'95*, volume 944 of *LNCS*. Springer, 1995.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proc. of the 18th IEEE Symposium on Foundation of Computer Science*, 1977.

[Pnu89] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, NATO Asi Series F: Computer And Systems Sciences. Springer, 1989.

[PP03] D. Perrin and J.-E. Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Academic Press, 2003.

[PR04a] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proc. of VMCAI'04*, volume 2937 of *LNCS*. Springer, 2004.

[PR04b] A. Podelski and A. Rybalchenko. Transition Invariants. In *Proc of LICS'04*. IEEE, 2004.

[PR05] A. Podelski and A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. In *Proc of POPL'05*. ACM Press, 2005.

[Pre29] M. Presburger. Über die Vollstandigkeit eines Gewissen Systems der Arithmetik. *Comptes Rendus du I Congrés des Pays Slaves*, 1929.

[PRSS02] A. Pnueli, Y. Rodeh, O. Strichmann, and M. Siegel. The Small Model Property: How Small Can It Be? *Information and Computation*, 178(1):279–293, 2002.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proc. of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001.

[PS00] A. Pnueli and E. Shahar. Liveness and Acceleration in Parameterized Verification. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-Counter Abstraction. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. of ISP'82*, volume 137 of *LNCS*. Springer, 1982.

[Rey02] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.

[RSL03] T. Reps, M. Sagiv, and A. Loginov. Finite Differencing of Logical Formulas with Applications to Program Analysis. In *Proc. of ESOP'03*, volume 2618 of *LNCS*. Springer, 2003.

[Ryb] A. Rybalchenko. ARMC: Abstraction Refinement Model Checker. URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`.

[Sai00] H. Saidi. Model Checking Guided Abstraction and Analysis. In *Proc. of SAS'00*, volume 1824 of *LNCS*. Springer, 2000.

[SB05] V. Schuppan and A. Biere. Liveness Checking as Safety Checking for Infinite State Spaces. In *Proc. of Infinity'05*, 2005.

[Sch02a] P. Schnoebelen. Verifying Lossy Channel Systems Has Nonprimitive Recursive Complexity. *Information Processing Letters*, 83(5):251–261, 2002.

[Sch02b] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[Sch06] M.I. Schwartzbach. Lecture Notes on Static Analysis, 2006. BRICS, Department of Computer Science, University of Aarhus, Denmark.

[Sha01] E. Shahar. *Tools and Techniques for Verifying Parameterized Systems*. PhD thesis, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2001.

[SMG97] A.P. Sistla, L. Miliades, and V. Gyuris. SMC: A Symmetry Based Model Checker for Verification of Liveness Properties. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[SP02] E. Shahar and A. Pnueli. Acceleration in Verification of Parameterized Tree Networks. Technical Report MCS02-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002.

[SRW02] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.

[SSM03] H. Seidl, T. Schwentick, and A. Muscholl. Numerical Document Queries. In *Proc. of PODS'03*. ACM Press, 2003.

[SSMH04] H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in Trees for Free. In *Proc. of ICALP'04*, volume 3142 of *LNCS*. Springer, 2004.

[TB73] B. A. Trakhtenbrot and Y. A. Barzdin. *Finite Automata: Behavior and Synthesis*. North-Holland, 1973.

[Tou01] T. Touili. Regular Model Checking Using Widening Techniques. *ENTCS*, 50, 2001.

[TV05] D. Tabakov and M.Y. Vardi. Experimental Evaluation of Classical Automata Constructions. In *Proc. of LPAR'05*, volume 3835 of *LNCS*. Springer, 2005.

[URM] The Uppsala regular model checking tool. URL: `http://www.regularmodelchecking.com/`.

[Val88] A. Valmari. *State Space Generation: Efficiency and Practicality.* PhD thesis, Tampere University of Technology, Tampere, Finland, 1988.

[Val98] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*. Springer, 1998.

[Ven99] A. Venet. Automatic Analysis of Pointer Aliasing for Untyped Programs. *Science of Computer Programming*, 35(2), 1999.

[Ven04] A. Venet. A Scalable Nonuniform Pointer Analysis for Embedded Programs. In *Proc. of SAS'04*, volume 3148 of *LNCS*. Springer, 2004.

[vN04] G. van Noord. FSA6.2, 2004. `http://odur.let.rug.nl/~vannoord/Fsa/`.

[VSVA04a] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively Learning to Verify Safety for FIFO Automata. In *Proc. of FSTTCS'04*, volume 3328 of *LNCS*. Springer, 2004.

[VSVA04b] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to Verify Safety Properties. In *Proc. of ICFEM'04*, volume 3308 of *LNCS*. Springer, 2004.

[VSVA05] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using Language Inference to Verify Omega-Regular Properties. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[VV05] A. Vardhan and M. Viswanathan. Learning to Verify Branching Time Properties. In *Proc. of ASE'05*. IEEE/ACM, 2005.

[vZGS78] J. von Zur Gathen and M. Sieveking. A Bound on Solutions of Linear Integer Equalities and Inequalities. *Proc. of the American Mathematical Society*, 1978.

[Wal96] I. Walukiewicz. Pushdown Processes: Games and Model Checking. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[WB95] P. Wolper and B. Boigelot. An Automata-Theoretic Approach to Presburger Arithmetic Constraints. In *Proc. of SAS'95*, volume 983 of *LNCS*. Springer, 1995.

[WB98] P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[WDHR06] M. De Wulf, L. Doyen, T.A. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[WKZ+06] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On Verifying Complex Properties Using Symbolic Shape Analysis. Technical Report MPI-I-2006-2-1, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2006.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Proc. of Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer, 1989.

[YKB02] T. Yavuz-Kahveci and T. Bultan. Automated Verification of Concurrent Linked Lists with Counters. In *Proc. of SAS'02*, volume 2477 of *LNCS*. Springer, 2002.