UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 419

Verifying Absence of ∞ Loops in Parameterized Protocols

MAYANK SAKSENA

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2008

ISSN 1651-6214
ISBN 978-91-554-7148-4
urn:nbn:se:uu:diva-8605

Dissertation presented at Uppsala University to be publicly examined in 2446, MIC, Polacksbacken, Uppsala, Friday, April 18, 2008 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Saksena, M. 2008. Verifying Absence of ∞ Loops in Parameterized Protocols. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 419. 80 pp. Uppsala. ISBN 978-91-554-7148-4.

The complex behavior of computer systems offers many challenges for *formal verification*. The analysis quickly becomes difficult as the number of participating processes increases.

A *parameterized system* is a family of systems parameterized on a number *n*, typically representing the number of participating processes. The *uniform verification problem* — to check whether a property holds for each instance — is an infinite-state problem. The automated analysis of parameterized and infinite-state systems has been the subject of research over the last 15–20 years. Much of the work has focused on safety properties. Progress in verification of liveness properties has been slow, as it is more difficult in general.

In this thesis, we consider verification of parameterized and infinite-state systems, with an emphasis on liveness, in the verification framework called *regular model checking (RMC)*. In RMC, states are represented as words, sets of states as regular expressions, and the transition relation as a regular relation.

We extend the automata-theoretic approach to RMC. We define a *specification logic* sufficiently strong to specify systems representable using RMC, and linear temporal logic properties of such systems, and provide an automatic translation from a specification into an analyzable model.

We develop *acceleration techniques* for RMC which allow more uniform and automatic verification than before, with greater power. Using these techniques, we succeed to verify safety and liveness properties of parameterized protocols from the literature.

We present a novel *reachability based* verification method for verification of liveness, in a general setting. We implement the method for RMC, with promising results.

Finally, we develop a framework for the verification of dynamic networks based on graph transformation, which generalizes the systems representable in RMC. In this framework we verify the latest version of the DYMO routing protocol, currently being considered for standardization by the IETF.

Keywords: formal methods, verification, model checking, infinite-state systems, regular model checking, liveness, graph transformation

Mayank Saksena, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

ISSN 1651-6214
ISBN 978-91-554-7148-4
urn:nbn:se:uu:diva-8605 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-8605)

To my family and friends.

List of Papers

This thesis is based on the following publications, which are referred to in the text as Papers A to D. The thesis contains extended versions of Papers A, B and D, and a reprint of Paper C.

A. Regular model checking for LTL(MSO).
Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena.
In R. Alur and D. Peled, editors, Proceedings 16th International Conference on Computer Aided Verification*, volume 3114 of* Lecture Notes in Computer Science*, pages 348–360. Springer Verlag, 2004.*

B. Proving liveness by backwards reachability.
Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena.
In C. Baier and H. Hermanns, editors, Proceedings CONCUR 2006, 17th International Conference on Concurrency Theory*, volume 4137 of* Lecture Notes in Computer Science*, pages 95–109. Springer Verlag, 2006.*

C. Systematic acceleration in regular model checking.
Bengt Jonsson and Mayank Saksena.
In W. Damm and H. Hermanns, editors, Proceedings 19th International Conference on Computer Aided Verification*, volume 4590 of* Lecture Notes in Computer Science*, pages 131–144. Springer Verlag, 2007.*

D. Graph grammar modeling and verification of ad hoc routing protocols.
Mayank Saksena, Oskar Wibling, and Bengt Jonsson.
In C. R. Ramakrishnan and J. Rehof, editors, Proceedings 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of* Lecture Notes in Computer Science*. Springer Verlag, 2008. To appear.*

Author order conventions.

Papers A–C have the authors in alphabetical order, which is what we normally do. Paper D — the only exception — has the students in alphabetical order and the professor last.

Contents

My Contribution

Roughly, Paper C is an improvement over Paper A in terms of verification results, and was originally motivated by Paper A. The work leading up to Paper C was in fact begun before Paper B, but was published afterwards. Paper B is an attempt to speed up the verification of liveness properties, by avoiding the computation of reachable loops, and as such was motivated by Paper A.

In Paper D we shift our focus from word shaped systems to graphs. Paper D started out as a case study of a routing protocol.

A. I participated in all parts of the work: discussions, implementation, experiments, proofs and writing. My main contributions were modeling and implementation.

For the version included in this thesis, I extended the theoretical part.

B. The original idea was Bengt Jonsson's. I played a major role in working it out and making the original formulation of the method stronger. I applied the method by hand to many examples, to motivate further work. Theorem 2 was my idea — it was necessary to verify key examples. I also implemented the method, and my implementation was used for the experiments.

I wrote the extended version included in this thesis (except the section about the alternating bit protocol).

C. This project started out being largely experimental. I led the implementation work throughout the project. I was the driving force behind the theory and implementation, guided by feedback from Bengt Jonsson.

D. Oskar Wibling and I collaborated closely on all the work presented in this paper. We developed the theory (including the proofs) together, as well as implementing and optimizing our verification tool. Oskar did most of the protocol modeling and led the implementation work. I devised proof strategies for our theory. Bengt Jonsson made many important suggestions, which had a significant impact on our work.

Other Publications

- **Insights to Angluin's learning.**
  Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena.
  In S. Etalle, S. Mukhopadhyay, and A. Roychoudhury, editors, Proceedings of the International Workshop on Software Verification and Validation (SVV 2003), volume 118 of Electronic Notes in Theoretical Computer Science, pages 3–18. Elsevier, 2005.

  Therese Berg and I did roughly the same share of work, supervised by Martin Leucker and Bengt Jonsson. This was my first project as a Ph.D. student, which is slightly off topic.

- **A survey of regular model checking.**
  Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena.
  In P. Gardner and N. Yoshida, editors, Proceedings CONCUR 2004, 15th International Conference on Concurrency Theory, volume 3170 of Lecture Notes in Computer Science, pages 35–48. Springer Verlag, 2004.

  Marcus Nilsson and Bengt Jonsson did most of the work. I wrote about extrapolation (essentially, Section 3.3 of the survey). This paper, and Marcus Nilsson's thesis [Nil05], are good sources for anyone curious about regular model checking.

Sammanfattning på svenska (Summary in Swedish)

> "Det finns inga signifikanta buggar i vår mjukvara som ett signifikant antal användare vill få åtgärdade."
>
> Bill Gates [Gat95]

Datorer blir allt mer en del av vårt vardagliga liv. Program körs i varje tänkbar apparat, och tillämpningarna har en enorm bredd; från fordon till pulsgeneratorer (pacemakers), till handel och telekommunikation... IT finns överallt. Däremot är det ytterst sällan vi kan garantera att ett program fungerar korrekt!

Om program som kontrollerar, exempelvis, en bils styrsystem fungerar felaktigt kan resultatet bli livsfarligt. Felaktiga program utgör ett enormt slöseri av resurser — även om det inte rör sig om säkerhetskritiska program — direkt, via förlorad funktionalitet, och indirekt, via arbetet som krävs för att rätta till felen.

Inom industrin idag så testas program, men utan att kunna ge korrekthetsgarantier. *Verifieringstekniker* är under utveckling inom den teoretiska datavetenskapen med syftet att ge ett definitivt svar på huruvida ett program fungerar korrekt.

Turing, Church, och Gödel grundade den teoretiska datavetenskapen på 30-talet, genom sina arbeten på att formalisera beräkningsbarhet (vad är en algoritm?), och deras resultat har fortfarande djupa implikationer för dagens vetenskap. Verifieringstekniker, och verktyg, har utvecklats sedan 70-talet. Det inom datavetenskapen prestigefyllda *Turingpriset* — vårt Nobelpris — gavs 2007 till Clarke, Emerson och Sifakis, för deras pionjärsarbete om verifieringstekniken *modellkontroll (model checking)*. Även om stora framsteg har gjorts sedan introduktionen på 70-talet, så utgör verifiering fortfarande ett svårt problem i allmänhet.

För att verifiera att ett program är korrekt, modellerar vi först programmet som ett *transitionssystem*, genom att representera alla *tillstånd* som programmet kan vara i, och *transitioner* mellan dessa tillstånd. Vi formulerar sedan en korrekthetsegenskap om programmets beteende, som ett villkor på transitionssystemets körningar. Vi kan alltså, något förenklat, betrakta programmets alla körningar som en (ibland oändlig) graf, och utföra vår korrekthetsanalys på denna graf. Målet för analysen

är att svara på om det finns någon körning som inte uppfyller korrekthetskravet — om det inte finns det, svarar vi att programmet är korrekt, annars konstruerar vi ett motexempel som visar varför programmet inte uppfyller kravet.

Program kan vara ytterst komplicerade, och det är en enorm utmaning att utveckla effektiva verifieringstekniker — program kan köras parallellt, eller distribuerat över ett nätverk, och utbyta information via meddelanden; och processer kan dynamiskt skapas och dödas. Många intressanta program uppvisar, tyvärr, ett enormt antal olika tillstånd, och är därmed svåra att analysera. Speciellt svårt är det att analysera parallella och distribuerade program, då antalet deltagande processer växer.

Avhandlingens innehåll i stora drag

Det finns ett behov för verifieringstekniker som kan analysera godtyckligt stora nätverk av processer. I denna avhandling förbättrar och utvecklar vi sådana tekniker.

Vi förbättrar verifieringsteknikerna inom ett specifikt modelleringsoch verifieringsramverk, kallat RMC, och lyckas verifiera icke-triviala egenskaper hos en stor mängd program för första gången.

Vi utvecklar också en helt ny teknik för verifiering av så kallade *framstegsegenskaper (progress/liveness)*, där man undrar om ett program till slut lyckas uppnå ett visst mål — ett typiskt exempel är att kontrollera huruvida en process som ber om tillgång till en resurs så småningom faktiskt får tillgång till den. Vi har implementerat tekniken inom RMC, och våra experiment är lovande. Det finns utrymme för intressant vidareutveckling på samma spår.

Slutligen har vi utvecklat ytterligare en verifieringsteknik, den här gången för *grafformade* system — ett typiskt exempel är ett nätverk av mobila noder som kommunicerar med varandra genom att skicka meddelanden (t.ex. studenter som kaotiskt rör sig runt i staden, med bärbara datorer, och chattar med varandra, eller forumsurfar ;). Vi gör en fallstudie på protokollet DYMO, som används för att möjliggöra kommunikation mellan trådlösa mobila noder då infrastruktur saknas, och verifierar att den senaste versionen av protokollet är korrekt. Detta protokoll håller på att granskas av IETF, för att eventuellt bli en Internetstandard.

Acknowledgments

My warmest thanks goes to my supervisor Bengt Jonsson. I have learned so much from working with you, not only about research. You are an excellent role model. Thank you for everything.

I also had the pleasure of working with Marcus Nilsson, who is a living example of that it is possible to master the field in both theory and practice. Thank you, my friend.

Without the support from my dear parents it would have been much harder to finish this thesis. Thanks for all the help.

Parosh Abdulla, I wish we would have worked more together. It was a good time. Thank you. Thanks also to Joachim Parrow, for a pleasant start of my graduate studies.

I also want to acknowledge the people at my department. You know who you are, and I will remember you always. Thanks in particular to my subset of the formal methods group (in alphabetical order) — Therese Berg, Johann Deneux, Olga Grinchtein, Frédéric Haziza, Noomene Ben Henda, Lisa Kaati, Pritha Mahata, Jonathan Mörndal, Marcus Nilsson, Julien d'Orso, Ahmed Rezine, Sven Sandberg, and Oskar Wibling. A special thanks to Oskar Wibling for helping me in the final stage of the writing, and for pleasant discussions. Thanks to Sven Sandberg for sharing his thesis scripts.

Finally, what would I do without my friends outside of the department? Kvarngärdet, Gränby, Fyris, Polacks, Norrlands, Uppsala... India. And all my dancing people. You are great. Thanks for being there.

Overview of Thesis

This thesis is organized as follows.

In Chapter 1 we motivate and give an overview of the research area.

Chapter 2 contains a presentation of problems we have worked with (as well as some interesting problems which we have not worked with).

In Chapter 3 we give a formal treatment of the program models which we have worked with in the papers, as a pleasant introduction for the reader. In Chapter 4 we give a formal treatment of safety and liveness properites, and present standard analysis methods. Chapters 3 and 4 should together give a student a good idea of the area.

In Chapter 5 we summarize the papers, and in Chapter 6 we present related work. We end the introduction by giving concluding remarks about our work, and a personal view on promising future research, in Chapter 7.

Papers A–D follow. Papers A, B and D are extended versions of the publications [AJN+04], [AJRS06], and [SWJ08]. Paper C is a (minimally edited) reprint of the publication [JS07], with an appendix.

1. Introduction

> "There are no significant bugs in our released software that any significant number of users want fixed."
>
> Bill Gates [Gat95]

Program Errors

Computer systems in domains such as telecommunications, industrial control, security, and even operating systems, are inherently complex. It is difficult to foresee and correctly handle all events to which a system must react. Therefore many programs behave incorrectly in particular situations. A large program is typically built by "interconnecting" modules, each of which provides functionality for solving a smaller problem. A module may be incorrectly documented or used, resulting in malfunction when used in an unhandled or unintended way.

Given the complexity of creating correct programs, whether they are implemented in hardware or software, it is not surprising that programs often have *bugs*; errors which cause the program to malfunction. Bugs may result in irritating behavior, such as your PC at home crashing, or life-threatening behavior, such as a pacemaker or car brake not functioning. Somewhere between "irritating" and "life-threatening" in importance, there is the cost to correct and replace a malfunctioning program, as well as potential damage caused to the user of the program, in terms of lost income and additional expenses.

By definition, *safety-critical systems*, such as pacemakers, and electronic controls for vehicles and traffic, power plants etc. must work correctly, since lives are at stake. For other systems, the tolerance for bugs is greater, yet it is desirable that they are correct; no-one (developer nor user) wants their resources to be wasted due to program errors.

There are many reasons why errors frequently occur in programs. Errors are often *logical*, i.e., due to an incorrect solution of the problem at hand. For example, the programmer may have had a particular case of the problem in mind and thereby missed special cases, resulting in an incomplete solution of the problem and infrequently occurring incorrect program behaviors. Errors may also be due to bad instructions (specifi-

cations), i.e., the programmer (attempted to) solve the wrong problem. As we will see, the techniques described in this thesis can be used to detect some errors of both types mentioned above.

The following famous bugs could likely have been avoided if more careful development and error detection methods had been used.

- In 1985–1987 a total of six patients, in USA and Canada, died or received serious injury due to overdoses caused by bugs in the radiation therapy machine *Therac-25* ; see e.g. [LT93].
- In 1994 a bug in the floating point division unit of Intel's Pentium processor was publically disclosed, which ultimately led to the recalling of those processors; see e.g. [Coe95].
- In 1996 the rocket *Ariane 5* self-destructed, caused by a variable having a value outside its expected range, resulting in a loss of about $370 million. The programmers did not foresee the problem, due to poor specifications [Dow97].

1.1 Verification

With the increasing amount of IT around us, we recognize the need for bug-free programs. But how can we avoid program errors? How do we find important errors? How can we ensure that a program is bug-free?

Correctness

One view of correctness is that of *functional correctness* — namely that for each possible input to the program, the correct output is produced. However, not all programs terminate; so-called *reactive systems* loop forever, reacting when input is received. To analyze also reactive systems we would thus like to generalize functional correctness to allow at least state dependent conditions — "on this input, when the program is in this state, then this output should be produced".

Since correctness is a rather vague concept, we require that a *specification* exists, which implicitly defines correctness; a specification is a description of how the program should and should not behave. Thus, a program is *correct* (with respect to a specification) if it satisfies the specification. We thus state the central problem studied in this thesis — the *verification problem*:

> Does a given *program* satisfy a given *specification*?

While, in practice, a specification can be ambiguous, or not exist at all, we need a precise specification to instantiate the verification problem. Hence, a specification stated in a precise language, such as a *logic*, is required before analysis can begin.

The verification problem can be applied at different levels of detail. Indeed, a specification can be seen as a high level program, and so the roles of "program" and "specification" in our formulation are better understood as descriptions of programs at lower and higher levels of detail.

The most straight-forward instantiation of the verification problem is software analysis, where the "program" is a mathematical model of the program under analysis, and the "specification" is a formal specification of the program. We can also analyze hardware; e.g., we can ask whether a circuit is correctly designed — then the "program" is a model of the circuit, and the "specification" is a description of what the circuit should compute. We can even analyze specifications, by asking whether they satisfy a set of *design requirements*; then the "program" is a specification, and the "specification" consists of the design requirements.

Note that even if we could solve the verification problem, the specification may not cover all bugs. Using verification we can only find bugs which violate a given specification. In other words, we can only answer whether a program is correct, if we have already defined what

correct means. However, this is the case for all program analysis. We can potentially solve the verification problem for any specification we can think of — hence, most critical bugs are amenable to our analysis.

Tackling the Verification Problem

A wide range of methods tackle the verification problem in different ways, from the practical but imprecise to the theoretical and precise (but sometimes impractical). On one end of the scale we have "common sense" and notions of good programming; testing, development and debugging tools and methodologies; and on the other end we have the topic of this thesis — formal verification.

The widely used meaning of *testing* a program is to examine if a number of computations (program execution scenarios) work as expected; for example, by inserting *assertions* in the code, or inspecting the value of data at certain program locations. Myers' classic work [Mye79] set the standards for systematic testing. Nonetheless, testing suffers from the following incompleteness issues:

1. It is often infeasible to test all program computations explicitly (unless the program is simple); thus the analysis is not complete.
2. Given that we cannot test all computations, how do we select good test cases?

The second problem of testing (see the list above) can be handled systematically by introducing different *coverage metrics*, which give meaning to how much program behavior has been tested. A test can be measured to have a certain percentage of coverage; the higher coverage, the more reliable is the test.

Perhaps surprisingly, the first problem is handled by *formal verification*, where the goal is to *prove* whether a program is correct. There is no contradiction here — it is indeed sometimes possible to guarantee correctness, even for large programs, by using mathematical methods!

In verification, we first construct a mathematical *model* of the program under analysis. Then, the model is analyzed for correctness. *Abstraction* is an essential part of the construction of a model — before applying verification techniques one should abstract away from detail (i.e., remove features not relevant for the analysis) to make the program smaller, and thus easier to analyze. If the model is verified to be correct, we conclude that the program is indeed correct. Otherwise, we have found an error in our model, which represents an error in the real program. In that case, we often want to improve the program, in a *debugging* phase, so that the found error does not occur, and repeat the above steps for the improved program.

The debugging phase is inherently difficult (or impossible?) to automate, as we have to fix a flaw in the design of the program — perhaps we

found a major flaw which cannot be solved by anything remotely similar to the program. This touches upon another approach to program errors — namely, to generate guaranteed correct code from specifications. While a beautiful thought, it does not allow us to analyze existing code, and has to tackle the difficult problem of generating efficient code.

Verification Techniques

Verification covers a large number of techniques. Large classes are: *static analysis*, *theorem proving*, and *model checking*. A short overview follows. For a broader overview of model checking and theorem proving, see e.g. [CWA+96], and for work closely related to this thesis, see Chapter 6.

Static analysis is the analysis of program code (as opposed to *dynamic analysis*, where the program executions are studied — although, the distinction is not always clear). Thus, the code is inspected in order to find flaws. Typical examples of what can be found by static analysis are memory leaks, dead code, uninitialized variables, indexing outside of arrays and dereferencing of null pointers. A well-known early tool for static analysis is Lint [Joh78]. State-of-the-art static analysis tools include Astrée [CCF+07] and TVLA [BLARS07], as well as a number of commercial tools.

In (interactive) theorem proving, the approach is semi-automatic (i.e., much human interaction is required), and the program analysis is done by an expert using a *theorem prover* (a program which assists the construction of proofs) to produce a correctness proof. Human interaction on the one hand enforces (and is helped by) human understanding of the analysis, but on the other hand it is very time consuming. (In contrast, model checking uses no human interaction during the analysis.) An important early contribution was LCF by Gordon et al. [GMW79]. State-of-the-art theorem provers include Coq [BC04, CH85], Isabelle [NPW02] and PVS [ORSSC98]. Famous applied results include (1) Nipkow et al. [NBS06] establishing the correctness of Hales' proof [Hal00] of the Kepler Conjecture; and (2) Gonthier's proof of the Four Color Theorem [Gon04].

Theoretical Limitations

Can we solve the verification problem, as we have informally defined it, for all programs and all specifications? Note that our verification methods are programs themselves — we write programs which analyze other programs. Our curiosity can thus be phrased:

> Does there exist a program which can *decide* whether any given program satisfies any given specification?

A famous *negative* answer to this question comes from *Turing's* proof that the *halting problem* is *undecidable* [Tur36]. Hence, we cannot solve the verification problem once and for all by writing a "universal verification program" — intuitively, we must make "program" and "specification" less general to obtain positive answers.

A consequence of Turing's result, and in fact any undecidability result concerning the verification problem (notably, [AK86]), is that we must restrict our ambitions of solving the verification problem to subclasses of programs (and specifications). We are thus left to either look for *decidable* subclasses, i.e. subclasses which we can guarantee answers for, or design verification methods which work well for a selection of interesting programs. In the latter case, we can only suggest that the method works well for "similar programs", rather than classifying a set of programs and specifications for which we can *guarantee* that it works.

Finite- and Infinite-State Programs

For the analysis of many classes of programs, typically communication and synchronization protocols, control programs, circuit designs, etc., we model a program as a *transition system* — at any given time the program is in a so-called *state* and the program can transition from one state to another via so-called *transitions*. We often refer to all possible states of a program collectively as the *state-space* of the program. Informally, we may look at a program under execution as executing an instruction, then pausing, then executing an instruction again, and so forth. A state is then a snapshot of the program during such a pause — essentially, the current data the program uses and a pointer to the next instruction (for each process). For example, the program variables and their values are part of a state.

Programs which have an infinite number of states are called *infinite-state programs* (and analogously we have *finite-state programs*). The source of infinity may be unbounded data domains, the number of processes, or unbounded data structures (such as queues or message buffers); there are infinitely many integers, reals etc.

But what do we really mean by infinite here? A pragmatic view is that everything is finite, and that infinity is an abstract mathematical concept. Certainly, computer memory is finite, and numbers in computers necessarily have finite range. However, we can often analyze a program with integers more successfully when treating them as being *arbitrarily large*, rather than finite (e.g., as 32 bit words).

Parameterized Systems

An important class of programs, in some sense between finite- and infinite-state, are the so-called *parameterized systems*, where an arbitrary number of processes n, given as a parameter, execute the same

program in parallel. The processes may communicate with each other, via shared memory or message passing. Many communication and synchronization protocols are parameterized systems — a number of participating processes each follow the same protocol to achieve some objective, as specified by the protocol.

While each instance of a parameterized system is finite-state, the collection of all instances is infinite-state, as the parameter is an arbitrarily large integer. The *uniform verification* of a parameterized system, where we ask whether each instance satisfies its specification, is therefore an infinite-state problem.

Examples of parameterized protocols, which we have also analyzed in this thesis, are *mutual exclusion* and *routing* protocols. A *mutual exclusion protocol* is a program executed by a number of processes, who compete for access to a resource. The purpose of a mutual-exclusion protocol is to ensure that the access to the resource is mutually exclusive — i.e., that at most one process has access to the resource at any time. Instances are lock-based solutions for accessing shared memory (e.g., reads and writes to a shared variable). Examples are the protocols of Burns [Bur81] (and Lamport [Lam86]), Dijkstra [Dij65] and Szymanski [Szy90], and Lamport's bakery protocol [Lam74, Lam79]. See also Lynch's book [Lyn96]. Correctness properties of interest for mutual exclusion protocols, other than *mutual exclusion*, are: *deadlock* (a.k.a *deadly embrace*) *freedom*, *starvation* (a.k.a. *lockout*) *freedom*, and whether the access to the resource is *First-Come-First-Serve*. A *deadlock* is a situation where two processes or more are waiting for each other to access the resource, with the result that no process obtains access [CES71]. A process *starves*, if it never accesses the resource despite wanting to do so.

A *routing protocol* is used to provide routing information in networks. More precisely, to provide information about which node (called the *next hop*) a message should be sent to, in order to eventually reach the desired destination. A current example of a routing protocol is the DYMO ad-hoc routing protocol [CP07], intended for networks whose structure can change frequently, e.g. due to nodes moving in and out of range of each other. Correctness properties of interest for routing protocols are: *routing loop freedom* — that there are no "loops" of next hop nodes (not including the destination), as this would cause a message to never reach its intended destination; and whether a sequence of next hops actually lead to the intended destination.

1.2 Model Checking

Model checking is a class of techniques for solving the verification problem for finite-state [CGP99] and infinite-state programs. Typically, the specification is formulated in a logic, and if the *model checker* (the analysis program conducting the verification) terminates, it outputs a "yes" if the specification is satisfied, or a "no" together with a *counterexample* — a computation showing a violation of the specification — if the specification is not satisfied.

The first model checking algorithms were introduced in the 80s independently by Clarke and Emerson [CE81, CES86], and Queille and Sifakis [QS82]. Only relatively small programs (having about a hundred thousand states), could be analyzed then. The introduction of model checking was recently recognized by the *ACM*, when Clarke, Emerson and Sifakis were given the 2007 *Turing award* for their pioneering work.

Another important contribution was the introduction of *symbolic model checking* (see, e.g., [McM93]), where the state-space exploration is done implicitly, by considering *symbolic representations* of potentially large sets of states. Symbolic model checking allowed the analysis of programs with trillions of states [BCM+92]. The underlying symbolic representation was BDDs [Bry86, Bry92]. In the late 90s, *bounded model checking* [BCCZ99, PBG05] emerged as a complementary technique, motivated by the success of *SAT solvers* [CM97]. In bounded model checking, all finite computations of a given length can be analyzed as a SAT problem. The analysis is typically repeated for longer computations, until a counterexample is found, or one runs out of memory or time.

Specifying Correctness

Two main classes of correctness properties are typically specified and analyzed: *safety properties* and *liveness properties* (see e.g. [Kin94]). The classification is attributed to Lamport [Lam77]. The concepts were later formalized by Alpern and Schneider [AS85], who also proved that any property is the intersection of a safety property and a liveness property.

Safety Properties

A safety property is of the form "something bad never happens (in any computation of the program)", where the "bad thing" typically represents a specific critical error. A common type of safety property says that "no *bad state* is visited" — meaning that bad states should not be visited in order for the program to function safely. For example, when checking mutual exclusion, we would classify any state where two or more processes simultaneously have access to the resource as bad. In

general, we can check whether a bad state can be visited in a computation by:

1. computing the states which are reachable by a sequence of transitions from an initial state, and checking whether they contain a bad state; or

2. computing the states which can reach a bad state by a sequence of transitions, and checking whether they contain an initial state.

We call (1) *forward reachability analysis*, and (2) *backward reachability analysis*.

Liveness Properties

A liveness property is essentially of the form "something good eventually happens (in every computation of the program)", usually under some conditions, i.e. "something good eventually happens whenever we have visited certain states". A liveness property thus typically represents some sort of progress requirement, such as "a process asking for a service eventually receives it", or "the computation eventually terminates".

For example, a liveness property can be specified as a (set of) pair(s) of states (s, t), with the meaning that "whenever state s is visited in a computation, then eventually t should be visited as well". A program model thus satisfies such a liveness property if all computations visit state t after visiting s. To answer whether the liveness property is satisfied we may:

1. exhaustively search for a computation which visits s but never visits t; or

2. compute the set of states which must visit t, and check whether s is included.

Option (1) is typically analyzed by a repeated search over the state-space, by so-called *repeated reachability analysis*. In Paper B we present a method based on option (2).

Model Checking Tools

At the time of writing, well-known (software) model checking tools include Blast [BHJM05], Magic [CCG+04], Slam [BR02], Spin [Hol97] and UppAal [LPY97]. UppAal is for the analysis of real-time systems, while the others are for "ordinary" systems, such as C/C++ programs. More recently, we have also Terminator [CPR06a, CPR06b]. There are also a number of tools for the verification of Java programs, such as Bandera [HDPR02], Java PathFinder [VHB+03], and jMoped [SSE05].

While many programming languages are similar from an analysis point of view, focusing on a widely used language has some benefits. First, it

allows the construction of tools working directly on "real" code, minimizing the need of manual modeling. Second, it gives access to a large amount of programs of industrial importance, which make up interesting benchmarks. All in all, working with "real" code is expected to speed up the emergence of verification tools in industry. The (non real-time) model checkers are typically able to analyze C programs with tens of thousands of lines of code, with a trade-off between precision and program size. Many practical and theoretical problems remain, such as scalability, pointer analysis, handling libraries (external code), and recursion.

State-of-the-Art Verification

There is a great incentive to incorporate formal verification into industrial-size software development, as an aid for bug finding in general. The emergence of various tools for model checking software (e.g., C/C++ and Java code) is promising, but we have yet to see large-scale use of these tools in the industry.

The nature of verification, i.e., an exact analysis which is sensitive to program size, seems to be more suitable for safety-critical systems. Such important programs must work correctly. Their core should be relatively simple, so that we can understand if and why they are correct. We should only trust that the program is error free if we can actually prove it. But, if the core is relatively simple, we should be able to automatically produce such a proof, much faster than a human expert would be able to...

2. Problem Statements

The complex behavior of computer systems[1] offers many challenges for formal verification. Programs may run in parallel on a single computer; or distributed over a network of computers, exchanging information through message passing; and processes may be dynamically created and destroyed during program execution. Computer systems typically exhibit a huge number of possible visited states and computations (often arbitrarily many). We understand that the analysis of parallel or distributed programs therefore quickly becomes difficult, or even infeasible, as the number of participating processes increases. While much of the work on model checking has been for finite-state systems, we recognize the need for methods able to analyze arbitrarily large networks of processes. In this thesis we try to improve the state-of-the-art verification techniques to better deal with the analysis of these programs.

The automated analysis of parameterized and infinite-state systems has been the subject of research over the last 15–20 years [AK86, GS92, EN96, WB98, KMM+01, ZP04, AJNS04]. One major approach has been to extend symbolic model checking techniques [BCM+90] to new classes of systems. Much of the work has been focusing on the verification of safety properties. Progress in the analysis of liveness properties has been slow, as this is a more difficult problem in general. For a more detailed account of related work, see Chapter 6.

Regular model checking was advocated by Kesten et al. [KMM+01] and by Boigelot and Wolper [WB98] as a framework for analyzing several classes of parameterized and infinite-state systems [WB98, KMM+01, AJNS04, Nil05]. This model checking framework is for systems whose states can be represented as finite words of arbitrary length over a finite alphabet. A set of states is symbolically represented as a regular set (i.e., as a regular expression) — hence the name "regular" model checking.

Systems suitably represented in this way include parameterized systems consisting of arbitrarily many finite-state processes, connected in a linear or ring-shaped topology, and systems operating on queues, stacks and other linear unbounded structures.

[1] By "computer system" we typically mean that more than one process is involved. It is a superficial distinction in formal verification, where a system is just a program.

Over the years, methods for computing the set of reachable states of a system, as well as the set of reachable loops (used to analyze liveness properties), have been developed for regular model checking [BJNT00, JN00, PS00, DLS01, AJNd02, AJNd03]. Computing the set of reachable states, as well as the transitive closure of the transition relation, is undecidable for these systems in general. However, decidability results for certain classes have been obtained [JN00].

The goal of regular model checking has always been to be a relatively general framework, and we should strive to support analysis of both safety and liveness properties. Within the framework, the analysis of safety properties is a computationally easier problem than liveness, and progress in the verification of liveness has been slow. Several specialized techniques exist which work for safety, but not liveness [ACJT00, ADHR07, ADR07] (see also Chapter 6). In this thesis we therefore aim to improve the state-of-the-art techniques for the verification of liveness properties, in particular within the regular model checking framework. We have considered the following specific problems.

Uniform Specification and Verification

Within the regular model checking framework, existing techniques for computing reachable states and reachable loops can in principle be used to verify both safety and liveness properties of parameterized system descriptions, but do not provide a convenient approach for checking arbitrary temporal logic properties of parameterized and infinite-state systems. Significant ingenuity is required in order to manually transform the verification of a temporal property of a parameterized system into a property of reachable states and reachable loops, in particular if the verification uses fairness properties that are parameterized on system components [BJNT00, PS00]. It would be desirable to have a framework, analogous to the automata-theoretic approach in finite-state model checking [Var07], where the property of verifying a temporal property is automatically transformed into a problem of checking emptiness for a Büchi automaton. Essentially, we ask ourselves:

> Can we design a specification language which is sufficiently expressive for the regular model checking framework, in which we can also specify arbitrary temporal properties, and automatically transform the system and property specifications into an analyzable verification model?

Another benefit of a uniform specification language is that of standardization — a common language allows several tools and methods to work on the same models, promoting competition, co-operation, and progress within the field.

Automatic Powerful Acceleration

A major problem when model checking parameterized and infinite-state systems is that the computation of fixed-points representing the set of reachable states or loops does not terminate in general, since there is no uniform bound on the number of transitions between reachable configurations. To make fixed-point computations terminate more frequently, so-called *acceleration* techniques have been developed, which calculate the effect of arbitrarily long sequences of transitions [BP96, BGWW97, ABJ98, ABJN99, BH99, PS00, ACABJ04, AJNS04, JS07] (sometimes called *meta transitions*).

Acceleration is undecidable in general. In practice, we can accelerate small relatively simple actions. Typically, encountered transition relations are large, and cannot directly be accelerated. A problem is, therefore, how to decompose a large transition relation into acceleratable subrelations. In other words, the research problem is essentially:

> Given an efficient acceleration technique for a class of transition relations, can we find a way to automatically and systematically apply the acceleration technique to arbitrarily large transition relations?

Light-Weight Verification of Liveness

The verification of liveness properties is more difficult than the verification of safety properties. Typically, verification of safety properties amounts to computing the set of reachable states, while liveness properties require computing the transitive closure of the transition relation.

The general approach for proving liveness involves finding auxiliary assertions associated with well-founded ranking functions and helpful directions (e.g., [MP84, MP96]). Finding such ranking functions is not easy, and automation requires techniques adapted to specific data domains.

This motivates the development of techniques for automatic verification, which avoid the computation of transitive closures, or other expensive computations. In other words:

> Can we design an automatic verification technique for liveness which is more light-weight than current approaches?

Such a technique which is complete would be a breakthrough. It is more reasonable to expect that we can find such an incomplete technique. We should then look for a technique which is powerful enough to motivate not being complete.

Verification of Graph Shaped Systems

The verification of network protocols has been one of the most important driving forces for the development of model checking technology. Most approaches (e.g., [Hol97, CGL92]) analyze finite-state models of protocols, but an increasing number of techniques are developed for analyzing parameterized or infinite-state models (see, e.g., [ACJT96, ZP04, AJNS04]).

There is a need for models which can represent networks with a potentially unbounded number of nodes, possibly with a dynamically changing topology. This covers a large class of systems, including protocols for wireless ad hoc networks, many distributed algorithms, security protocols, telephone system services, etc. The framework of regular model checking, for example, cannot handle such systems in a straight-forward way, as they would have to be encoded into words. We thus ask:

> Can we construct an efficient verification framework suitable for the verification of network protocols over arbitrary topologies, and in general, graph shaped systems?

□

Before presenting our contributions, we give an introduction to the program models we have used, focusing on the regular model checking framework, and to the verification of safety and liveness properties in general, in Chapters 3 and 4.

3. Program Models

In this thesis, we have used a number of different models for the analysis of programs. Each model is suitable for modeling a specific type of program, and for each model we will present a particular analysis method (as we will see in the Papers). All models are *transition systems*; they represent a program as a "state machine" consisting of a set of *states* which the program can be in, and *transitions* between these states. We use the standard definition of a transition system.

Definition 3.1 (Transition System)
A *transition system* is a tuple $\mathcal{P} = \langle Q, I, \longrightarrow \rangle$, where

- Q is a set of *states*,
- $I \subseteq Q$ is a set of *initial states*, and
- $\longrightarrow \subseteq Q \times Q$ is a *transition relation*.

If $(s,t) \in \longrightarrow$, also written $s \longrightarrow t$, it means that the program can transition (i.e., change state) from s to t.

A *computation* of $\mathcal{P}$ from a state $s \in Q$ is a (possibly infinite) sequence of states $\sigma = s_0\ s_1\ s_2\ \cdots$ where $s_0 = s$, and $s_{j-1} \longrightarrow s_j$ for each $j \geq 1$ (and j less than the length of σ if the computation is finite). We assume that the computation is *initialized* by default, i.e., that $s \in I$, unless otherwise stated.

The transition relation is often given as a union of *actions*. An *action* is simply a relation $\alpha \subseteq \longrightarrow$. To specify that $(s,t) \in \alpha$, we write $s \xrightarrow{\alpha} t$, and say that $\mathcal{P}$ from state s can *execute* (or *take*) action α and reach state t. Action α is *enabled* in state s if there is some state t such that $s \xrightarrow{\alpha} t$. The set of states in which the action α is enabled is denoted $En(\alpha)$. □

Running Example
To demonstrate our program models, we will use a (toy) example — the *token ring* protocol. A token is passed around between a fixed number n of participating processes. The intention is that the process which holds the token has exclusive rights to some resource. The processes are indexed from 0 to $n-1$, and initially the token is at process 0, say.

The token can be passed from process i to its higher-indexed neighbor, process $(i+1) \bmod n$. We can model the system using a word of length n. The letter at position i, denoted $w(i)$, then models the local state of process i. The local state of a process is either $\top$, meaning that this process has a token, or $\bot$, otherwise.

A transition system model $\langle Q, I, \longrightarrow \rangle$ of this protocol, could thus be the following, where the indices increase to the right and the leftmost process has index 0.

- $Q = \{\top, \bot\}^n$ — the set of states are all words of length n over the alphabet $\{\top, \bot\}$
- $I = \{\top \cdot \bot^{n-1}\}$ — the initial state is the word where a token is at the leftmost process, and the other $n-1$ processes have no token
- $\longrightarrow = \{\langle w(i) = \top\,;\, w(i) = \bot, w((i{+}1) \bmod n) = \top \rangle \mid i = 0..n{-}1\}$, meaning that any process i which has the token can pass the token to its right neighbor.

The predicates $w(j) = \top$ and $w(j) = \bot$ above mean that process j has, respectively has not, got a token.

3.1 Fair Transition Systems

In Papers A, B and C, we use the standard concept of *(weak) fairness*, a.k.a. *justice* [MP96]. Fairness allows us to model natural progress requirements. For example, it is obvious that a process will eventually take an action which remains enabled; at least if all other processes run on different processors. Without assuming fairness, one process may repeatedly act in a computation, even though another process could act independently — which is unrealistic if the processes run on different processors, and "unfair" if they run on the same processor. When analyzing liveness properties, fairness conditions are used to discard such unrealistic or unfair behavior.

We now extend our basic definition of computation to respect certain fairness conditions.

Definition 3.2 (Fair Transition System)
A *fair transition system* is a tuple $\mathcal{P} = \langle Q, I, \longrightarrow, \mathcal{A} \rangle$, where

- $\langle Q, I, \longrightarrow \rangle$ is a transition system, and
- $\mathcal{A}$ is a set of actions, denoted *fair actions*.

A *fair computation* of $\mathcal{P}$ from a state $s \in Q$ is an infinite computation $\sigma = s_0\ s_1\ s_2\ \cdots$ from s which satisfies the following *fairness condition*: for each $\alpha \in \mathcal{A}$, there are infinitely many $i \geq 0$ where either $s_i \xrightarrow{\alpha} s_{i+1}$ or $s_i \notin En(\alpha)$. □

In our definition of fairness, the indices i represent "opportunities to act". Thus, we require that a fair action should be executed, or blocked (not enabled), infinitely often (intuitively, "when it gets an opportunity to act"). In particular, it follows that a computation in which some fair action is enabled in each state, but never executed, is not a fair computation.

Why Infinite Computations?

Note that we defined fair computations to be infinite. The reason is that finite computations are not sufficient to analyze liveness properties of programs which do not terminate, such as reactive systems.

An example of a property which requires infinite computations is the absence of *livelocks* — a situation where two or more processes loop repeatedly, stopping each other from progressing. Intuitively, this situation is similar to what can happen if two people meet in a corridor, trying to pass each other, all the time choosing the same side.

We can also model finite computations as infinite, by including an *idle* action which does nothing; we would then model a finite computation as having an infinite suffix of the last state s of the finite computation, i.e.,

s forever idles (transitions to itself on the idle action). Hence, without loss of generality, we consider only infinite computations.

Running Example

Now suppose that we add the *idle* action, which equals the identity relation $\{\langle q, q\rangle \mid q \in Q\} \subseteq Q \times Q$, to the transition relation.

First, if $\mathcal{A} = \emptyset$, then the fair computations of the program are all computations where the token is passed an arbitrary number of times, with an arbitrary number of idling steps in between. For example, the token need not be passed at all — the idle action can repeatedly be executed.

Now suppose that we instead let the token passing action be fair. Since this action is always enabled, the fair computations now involve an infinite number of executions of the token passing action. Intuitively, this corresponds to the assumption that a process cannot forever hold the token — it must eventually pass it on to its neighbor.

Büchi Acceptance Conditions

A *Büchi automaton* [Büc62, Tho90] is an acceptor of infinite words, typically used to represent properties of infinite computations. A Büchi automaton is a finite automaton with so-called *Büchi acceptance conditions*, used to specify which infinite words are accepted.

Here, and in Paper A, we use Büchi acceptance conditions to describe the computations of a transition system.

Definition 3.3 (Büchi Transition System)
A *Büchi transition system* is a tuple $\mathcal{P} = \langle Q, I, \longrightarrow, F\rangle$, where

- $\langle Q, I, \longrightarrow\rangle$ is a transition system, and
- F is a set of *accepting states*.

A *computation* of $\mathcal{P}$ from a state $s \in Q$ is an infinite computation $\sigma = s_0\, s_1\, s_2\, \cdots$ of $\langle Q, I, \longrightarrow\rangle$ from s which satisfies the following *Büchi acceptance condition*: some accepting state is infinitely often visited; i.e., there exists $s \in F$ such that $s_i = s$ for infinitely many $i \geq 0$. □

Running Example

Let us see what happens if we let $F = I$. By definition, then the computations of the system have to visit I infinitely often. This can be achieved by eventually visiting I and then idling forever, or passing the token around infinitely often.

3.2 Regular Transition Systems

Papers A and C are conducted within the verification framework called *Regular Model Checking (RMC)* [Nil05, AJNS04], where *regular relations* are used to model programs. We here use the standard automata theoretic definition of "regular", see e.g. [HMU07].

Regular model checking is a framework for the symbolic verification of parameterized and infinite-state systems [WB98, BJNT00, KMM$^+$01, Mai01, BLW03]. The program is modeled as a regular transition system; thus a configuration should be represented as a word over a finite alphabet. Regular transition systems are therefore suitable for modeling linear structures, such as a vector of finite-state processes, or stacks and queues.

If the transition relation is regular and *length-preserving*, meaning that related configurations have the same length, we can efficiently represent it using a finite automaton over the alphabet $\Sigma \times \Sigma$ — a so-called *transducer*. For example, a regular transition relation $\longrightarrow = \{(a_1 \cdots a_n, b_1 \cdots b_n)\}$ can be represented by a transducer which accepts the language $\{(a_1, b_1) \cdots (a_n, b_n) \mid (a_1 \cdots a_n, b_1 \cdots b_n) \in \longrightarrow\}$.

The limitation that transition relations should be length-preserving sometimes makes modeling dynamic structures difficult (or impossible). We can however represent that an arbitrarily large amount of space is "pre-allocated" by including an arbitrary number of "empty slots", using so-called *padding symbols*.

Preliminaries

An *alphabet* is a finite set of symbols, denoted Σ. We use $\Sigma^\star$ to denote the set of *finite words* over Σ.

A relation $\mathcal{R}$ on $\Sigma^\star$ is *length-preserving* if w and w' have the same length whenever $(w, w') \in \mathcal{R}$.

A relation $\mathcal{R}$ on $\Sigma^\star$ is *regular* if it is length-preserving and the set $\{(a_1, b_1) \cdots (a_n, b_n) \mid (a_1 \cdots a_n, b_1 \cdots b_n) \in \mathcal{R}\}$ is regular — i.e., is accepted by a finite automaton with alphabet $\Sigma \times \Sigma$.

Given the definition of regular relation, the program models of RMC are straight-forward to define — we simply require some relations to be regular.

Definition 3.4 (Regular Transition System)
A *regular transition system (over alphabet Σ)* is a transition system $\mathcal{P} = \langle Q, I, \longrightarrow \rangle$, where

- $Q \subseteq \Sigma^\star$ is a regular set of *configurations*,
- I is a regular set of *initial configurations*, and

- $\longrightarrow \subseteq Q \times Q$ is a regular relation. □

We typically use "configuration" to denote a state which represents a "global state" (e.g., representing more than one process). Note that a configuration is a word over Σ. Often a word $w \in Q$ represents a vector of finite-state processes, each having their "local state" in Σ.

Definition 3.5 (Büchi Regular Transition System)
A *Büchi regular transition system* (over alphabet Σ) is a Büchi transition system $\mathcal{P} = \langle Q, I, \longrightarrow, F \rangle$, where

- $\langle Q, I, \longrightarrow \rangle$ is a regular transition system (over Σ). □

In Paper A, we let F be a relation instead, which allows us to specify *accepting transitions* — i.e., we can say that a pair of states should be visited infinitely often. That is a convenient formulation, which is however equivalent, since we can encode accepting transitions as accepting states by adding extra state.

Running Example
Let us now model the token passing protocol as a parameterized system, by letting the number of processes n be a parameter. Thus, every value of n defines one transition system, like the one we saw before.

A nice thing is now that we can model all instances of the protocol as a single regular transition system. As we saw before, the states of a system instance with n processes, can be represented as words of length n. As a symbolic representation of all instances, we use regular expressions. A comparison with the transition system model shows that we simply generalize from n to $\star$.

- $Q = \{\top, \bot\}^\star$ — the set of states are all words of arbitrary length over the alphabet $\{\top, \bot\}$
- $I = \top \cdot \bot^\star$ — the set of initial states contains all words where a token is at the leftmost process, and the other processes have no token
- $\longrightarrow = \{\langle w \cdot \top \cdot \bot \cdot w' ; w \cdot \bot \cdot \top \cdot w' \rangle\} \cup \{\langle \bot \cdot w \cdot \top ; \top \cdot w \cdot \bot \rangle\}$ where $w, w' \in Q$ are arbitrary words.

The first part of $\longrightarrow$ describes an action where a process, which is not the rightmost one, passes the token to its right neighbor. The second part describes what happens when the rightmost process passes the token, namely that the token reaches the leftmost process.

4. Correctness

In this thesis, we are concerned with the analysis of correctness of programs. In Chapter 3 we described our program models (as transition systems). Here we will clarify what we mean by correctness, and what type of analyses we actually make.

Specifying Correctness Properties

We formally define a *property*, φ, to be a (usually infinite) set of computations. We say that a computation σ satisfies a property φ, written $\sigma \models \varphi$, if $\sigma \in \varphi$. Thus, a property is specified as the set of all satisfying computations. A transition system satisifies a property if every computation of the transition system satisfies the property. If a computation or transition system does not satisfy a property, we say that it *violates* the property. We call a computation which violates the property a *counterexample* to the property.

Two classes of properties are typically distinguished: *safety properties* and *liveness properties* [Lam77, AS85, Kin94]. It can be shown that any property is the intersection of a safety property and a liveness property [AS85].

4.1 Safety Properties

A safety property is of the form "something bad never happens (in any computation of the program)", where the "bad thing" typically represents a critical error. Following the style of Alpern and Schneider [AS85, Sis85], the intuition is that the "error is irremediable", i.e. that any computation which extends a computation violating the property also violates the property. Hence a violating computation has some *finite prefix* leading to the error. This latter formulation is the basis of the formal definition of a safety property in [AS85], which we will use here. Below, we use the notation $\sigma[i..j]$ for the finite *subcomputation* $s_i \cdot \ldots \cdot s_j$ of $\sigma = s_0\ s_1\ s_2\ \cdots$.

Definition 4.1 (Safety Property)
A *safety property* φ is a property which for all computations σ satisfies:

$$\sigma \models \varphi \quad \Longleftrightarrow \quad \forall i \geq 0\ \ \exists\, \sigma' :\ \sigma[0..i] \cdot \sigma' \models \varphi\ .$$

□

The definition says that a computation σ satisfies the property if and only if every prefix of σ has some extension which satisfies the property.

Note that if we negate both sides we obtain the equivalent statement "σ violates φ if and only if there is a prefix of σ such that all its extensions violate φ" — just as the intuitive formulation above.

It follows that a system violates a safety property if and only if there exists a finite prefix of some computation σ, such that all extending computations $\sigma \cdot \sigma'$ violate the property. Thus, in that case, there exists a finite prefix which, intuitively, witnesses an irremediable error.

In light of the above, we understand that a counterexample to a safety property is characterized by a finite computation. Equivalently, we could therefore specify a safety property as the set of prefixes which characterize its counterexamples.

Examples of Safety Properties

Consider a set of program states S and a property of form "a process never visits a state in S". This is a safety property, where the "bad thing" is visiting S. If a computation visits S, then whatever extends the prefix leading up to S also violates the property. Similarly, if there is a prefix for which all extensions violate the property, then this prefix must (eventually) lead up to S. If we add a condition, such as "a process never visits S after visiting T", where T is another set of states, we get another safety property, where the "bad thing" is visiting S after visiting T. A computation where the process has visited S and thereafter never sees T, or never sees S at all, satisfies the property. A computation where the process has seen S, but afterwards visits T, violates the property.

4.2 Liveness Properties

A liveness property is of the form "something good eventually happens (in every computation of the program)". Often the "good thing" should happen under some conditions, i.e. "something good eventually happens provided that we have visited certain states". A liveness property typically represents some sort of progress requirement, such as "a process asking for a service eventually receives it", or "the computation eventually terminates".

Again using the definitions of [AS85], the defining characteristic for liveness is that the "good thing" can always be postponed, i.e. that any partial computation is allowed before the "good thing" happens. Thus, if a computation σ' satisfies a liveness property, then any finite computation which has σ' as suffix should also satisfy the property.

Below, we use the notation $Q^\star$ for the set of all finite sequences of states.

Definition 4.2 (Liveness Property)
A *liveness property* φ is a property satisfying:

$$\forall \sigma \in Q^\star \;\; \exists \sigma' : \;\; \sigma \cdot \sigma' \models \varphi \; .$$

□

The definition says that any finite computation must have an extension which satisfies the liveness property, following the intuitive formulation above.

Examples of Liveness Properties

Consider a liveness property of form "a process eventually receives a service", and a finite computation σ where the process has not yet received the service. According to the definition, any extension of σ where the process eventually receives the service, no matter how long it has to wait, satisfies the property. A computation where the process never receives the service violates the property.

If we add a condition, such as "a process asking for a service eventually receives it", we get the following. Any extension of σ where the process has asked for the service and eventually receives the service, no matter how long it has to wait, satisfies the property. Computations where the process never asks for the service also satisfy the property. Finally, a computation where the process has asked for the service, but never receives it, violates the property.

4.3 Analysis

In the *automata-theoretic* approach (see, e.g., [VW86, Var91] and Paper A), the system and property are given as (or translated to) Büchi transition systems (Definition 3.3), allowing the analysis of arbitrary correctness properties. In Paper A we describe the translation for regular transition systems. Here we instead look directly at safety and liveness properties.

4.3.1 Analyzing Safety

How can we check whether a transition system satisfies a given safety property φ? To check whether a safety property with an infinite set of counterexamples is satisfied, we need a finite representation of them. In general, an *observer* is used — a transition system which keeps track of whether a counterexample has been seen. In the automata-theoretic approach, we use automata for this purpose.

Given a program and a safety property φ we can check whether the program satisfies the property by:

1. modeling the program as a transition system $\mathcal{P}$;
2. specifying the computations which violate φ as an automaton $\mathcal{A}_{\neg\varphi}$ which accepts exactly all the prefixes which characterize the counterexamples to the safety property;
3. constructing the *synchronized product* $\mathcal{P} \times \mathcal{A}_{\neg\varphi}$;
4. checking whether there is a computation of $\mathcal{P} \times \mathcal{A}_{\neg\varphi}$ which visits an *accepting state* of $\mathcal{A}_{\neg\varphi}$.

In the last step, we essentially run $\mathcal{P}$ and $\mathcal{A}_{\neg\varphi}$ in parallell, and answer "not satisfied" if we reach an accepting state of $\mathcal{A}_{\neg\varphi}$. It is sufficient to compute the states of $\mathcal{P} \times \mathcal{A}_{\neg\varphi}$ which are *reachable*. Below, we formalize the fixed-point analysis.

Definition 4.3 (Post- and Pre-Images)
The *post-$\star$-image* of a set of states $S \subseteq Q$ with respect to a set of actions $\mathcal{A}$, denoted $\mathit{Post}^\star(\mathcal{A}, S)$, is the set of states t such that there exists a sequence of transitions $s \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} t$ where $s \in S$, and $\alpha_j \in \mathcal{A}$ for each $j \in [1, n]$. The *post-image*, denoted $\mathit{Post}(\mathcal{A}, S)$, is defined as the post-$\star$-image, except that the sequence is of length 1 (i.e., $n = 1$).

The *pre-$\star$-image*, $\mathit{Pre}^\star(\mathcal{A}, S)$, is the set of states t such that there exists a sequence of transitions $t \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} s$ where $s \in S$, and $\alpha_j \in \mathcal{A}$ for each $j \in [1, n]$. The *pre-image*, $\mathit{Pre}(\mathcal{A}, S)$, is defined analogously.

If t is in $\mathit{Post}^\star(\mathcal{A}, S)$, we say that t is *(forwards) reachable* from S (with respect to $\mathcal{A}$). If t is in $\mathit{Post}(\mathcal{A}, S)$, we call t an (*immediate*)

successor of S. We analogously define *predecessor* and *backward reachability*, and extend our image definitions to allow single states and actions as parameters. □

Definition 4.4 (Reachability)
A set of states $T \subseteq Q$ is *(forwards) reachable* from a set of states $S \subseteq Q$ with respect to a set of actions $\mathcal{A}$ if $T \cap Post^{\star}(\mathcal{A}, S) \neq \emptyset$.

A set of states $T \subseteq Q$ is *backwards reachable* from a set of states $S \subseteq Q$ with respect to a set of actions $\mathcal{A}$ if $T \cap Pre^{\star}(\mathcal{A}, S) \neq \emptyset$.

By default, if no actions $\mathcal{A}$ are mentioned, we assume that reachability is with respect to the entire transition relation ($\longrightarrow$). □

Definition 4.5 (Synchronized Product)
Given a transition system $\mathcal{P} = \langle Q, I, \longrightarrow \rangle$ and an automaton $\mathcal{A} = \langle Q_{\mathcal{A}}, I_{\mathcal{A}}, \longrightarrow_{\mathcal{A}}, F_{\mathcal{A}} \rangle$ with accepting states $F_{\mathcal{A}} \subseteq Q_{\mathcal{A}}$.

The *synchronized product* of $\mathcal{P}$ and $\mathcal{A}$ is the automaton

$$\mathcal{P} \times \mathcal{A} = \langle Q \times Q_{\mathcal{A}}, I \times I_{\mathcal{A}}, \longrightarrow_{\times}, Q \times F_{\mathcal{A}} \rangle$$

where $(s,t) \longrightarrow_{\times} (s',t')$ if and only if $s \longrightarrow s'$ and $t \xrightarrow{s'}_{\mathcal{A}} t'$. □

Definition 4.6 (Safety by Reachability)
Given a transition system $\mathcal{P} = \langle Q, I, \longrightarrow \rangle$ and a safety property φ. Suppose that $\mathcal{A}_{\neg\varphi} = \langle Q_{\neg\varphi}, I_{\neg\varphi}, \longrightarrow_{\neg\varphi}, F_{\neg\varphi} \rangle$ is an automaton accepting precisely a set of prefixes which characterize all counterexamples to the property. Construct the synchronized product $\mathcal{P} \times \mathcal{A}_{\neg\varphi}$. The *violating states* of the synchronized product are $Q \times F_{\neg\varphi}$.

$\mathcal{P}$ satisfies φ if and only if the set of violating states are not forwards reachable from $I \times I_{\neg\varphi}$. Equivalently, $\mathcal{P}$ satisfies φ if and only if $I \times I_{\neg\varphi}$ is not backwards reachable from any violating state. □

4.3.2 Analyzing Liveness

Given a fair transition system $\mathcal{P} = \langle Q, I, \longrightarrow, \mathcal{A} \rangle$ and a liveness property φ. Following the automata-theoretic approach, we look for counterexamples to the property; namely any computation of the transition system for which the "good thing" never happens. Thus, we should augment the transition system $\mathcal{P}$ so that $\neg\varphi$ is enforced, and check whether the resulting system actually has a computation.

In Paper A (see also [VW86]), we show a reduction from a fair transition system $\mathcal{P}$ and $\neg\varphi$ to a Büchi transition system by encoding the fairness conditions as Büchi acceptance conditions.

Infinite-state systems can have computations which do not visit any state twice. Hence, not all infinite-state systems are representable as Büchi transition systems.

Given a Büchi transition system, we can find counterexamples to the liveness property by using a technique based on so-called *loop-finding*.

Below, we formalize the technique, which essentially conducts a repeated search over the states.

Definition 4.7 (Repeated Reachability)
The *reflexive-transitive closure* of a transition relation $\longrightarrow$ is the relation $\{(s,t) \mid t \in \mathit{Post}^\star(\longrightarrow, s)\}$.

A state t is *repeatedly reachable* from a state s with respect to a transition relation $\longrightarrow$ if $t \in \mathit{Post}^\star(\longrightarrow, s)$ and there exists some t' such that $t' \in \mathit{Post}^\star(\longrightarrow, t)$ and $t \in \mathit{Post}^\star(\longrightarrow, t')$. □

The definition says that a state t is repeatedly reachable from s if it can be visited repeatedly in a "loop" in a computation from s.

We see that the reflexive-transitive closure of a transition relation allows us to check repeated reachability.

Definition 4.8 (Liveness by Loop-Finding)
Given a fair transition system $\mathcal{P}$ and a liveness property φ.

Given a Büchi transition system $\mathcal{P}_\times = \langle Q_\times, I_\times, \longrightarrow_\times, F_\times \rangle$ which exactly represents all computations of $\mathcal{P}$ which violate φ.[1]

The following technique, called *repeated reachability analysis*, or *loop-finding*, simply computes a representation of all "loop-shaped" computations of $\mathcal{P}_\times$ — the "reachable fair loops".

1. Compute the reachable states $\mathit{Inv} = \mathit{Post}^\star(\longrightarrow_\times, I_\times)$.
2. Compute the transitive closure of $\longrightarrow_\times$ starting from $F_\times$:
$$\longrightarrow_{RR} = \{(s,t) \mid s \in \mathit{Inv} \cap F_\times \text{ and } t \in \mathit{Post}^\star(\longrightarrow_\times, s)\} .$$
3. The set of states $RR = \{s \mid (s,s) \in \longrightarrow_{RR}\}$ are repeatedly reachable from some initial state.

If $RR \neq \emptyset$ then $\mathcal{A}_\times$ has a fair computation, which by construction is a counterexample to the property. □

Remark. The technique is *sound* for Büchi transition systems; i.e., if a "reachable loop" is found, there exists a violating computation, and the property is false.

The technique is (obviously) *complete* for Büchi transition systems where all computations are loop-shaped. This includes, e.g., finite-state and regular transition systems.

[1] The construction is not central here. Suffice it to say that the cross product used for safety works, except we have to make sure that both Büchi conditions are satisfied componentwise, by using an "alternating" construction.

It is not complete for Büchi transition systems in general, as we note that a state can be infinitely often visited in a computation which is not "loop-shaped" — hence, computations which are not "loop-shaped" may exist.

5. Summary of Papers

This chapter contains a summary of the papers included in this thesis.

A. *Regular model checking for LTL(MSO).*

An important component of automated analysis is the modeling and specification of the system and property, and their translation into a verification problem.

The manual translation of a fair transition system and a property into an analyzable Büchi transition system is non-trivial and error-prone, in particular for regular model checking.

In the automata-theoretic approach [Var91] one checks whether a property φ of a program $\mathcal{P}$ is true, by constructing a Büchi automaton which accepts all *incorrect* computations of $\mathcal{P}$, i.e., every computation which violates φ. The point of the automata-theoretic approach is to reduce the checking of any correctness property to one problem, namely checking whether a corresponding Büchi automaton has an empty language. The main contribution of Paper A is to extend the automata-theoretic approach to regular transition systems.

A two-dimensional specification logic, called $LTL(MSO)$, is defined for the specification of programs and correctness properties. The *space* dimension is used to model system configurations, and the *time* dimension is used to express linear time properties.

We present an automatic procedure for translating any (restricted) $LTL(MSO)$ formula φ into a *Büchi regular transition system* (see Definition 3.5) whose computations are exactly the set of all models of the formula.

A program $\mathcal{P}$ and a (negated) property φ, both specified in $LTL(MSO)$, can thus be translated into a Büchi transition system accepting precisely all computations of $\mathcal{P}$ which violate φ.

Marcus Nilsson and I implemented the translation for our *regular model checking* framework (see, e.g., [Nil05]) and automatically analyzed a number of parameterized mutual-exclusion protocols from the literature.

B. *Proving liveness by backwards reachability.*

The analysis of liveness properties is more difficult than the analysis of safety properties. Established complete techniques are based on finding a well-founded *ranking function* [MP96], or computing the transitive closure of the transition relation [VW86] (see also Paper A).

We introduce a technique for proving liveness properties based on reachability, thus avoiding the potentially expensive computation of the transitive closure of the transition relation.

The technique is expected to scale better than techniques based on the computation of reachable loops (see Chapter 4), but it is not meaningful for the analysis of safety properties.

The reduction of liveness to fair termination is a standard task [Var91]. In this work, we assume that the liveness property has been reduced to the problem of fair termination.

A liveness property of form "whenever S is visited, T will eventually be visited", where S and T are sets of states, is analyzed by computing a set of states which are *terminating* — here, *termination* means to reach T. If we find that all states in S are terminating, then the original liveness property is true.

Starting out with a set of states known to terminate (T in our example), we compute a larger set of states which is guaranteed to also terminate (i.e., which eventually must visit T), using backward reachability analysis with a special transition relation.

Even though the technique is incomplete, it potentially simplifies further analysis (if such is needed) by enlarging the set of states known to terminate. For example, we may find out that some set of states V is terminating. The original termination problem can now be simplified to checking whether $S \setminus V$ leads to $T \cup V$.

Since the technique computes a set of states T guaranteed to terminate, and is incomplete, we can not say anything about states not in T — they may or may not terminate. However, the technique is powerful enough to verify liveness of many infinite-state programs, including the parameterized protocols considered in Paper A.

We have implemented the technique in our regular model checking framework, and automatically analyzed the programs considered in Paper A. Additionally, we used the technique to verify some programs, for which we did not have backward reachability analysis tools, by hand.

C. *Systematic acceleration in regular model checking.*

The computation of reachable states and loops (see Definitions 4.4 and 4.7) of infinite-state systems is a major problem.

Fixed-point computations used to compute reachable states and loops do not terminate in general for infinite-state systems, as the number of transitions between reachable states and loops can be infinite. To make the computations terminate more often, acceleration techniques have been developed (see Chapter 2).

In this paper, we present a technique for automatically accelerating all actions of a certain kind, for the analysis of regular transition systems. The actions considered are so-called *unary separable* actions, because we can accelerate them efficiently [ABJN99, JN00, PS00], and they occur frequently in parameterized programs. The acceleration technique can, in principle, be extended to any class of actions which we can accelerate efficiently.

The technique improves upon previous work (e.g., Paper A) in two ways: (1) it allows us to automatically accelerate certain actions, which previously were accelerated manually (they were included in the system model); and (2) a larger number of actions are accelerated, which therefore improves the power of our fixed-point analysis.

Using the techniques of this paper, we were able to verify liveness properties of certain parameterized protocols for the first time.

D. *Graph grammar modeling and verification of ad hoc routing protocols.*

In this paper, we consider a more general class of systems (although we only analyze safety properties).

We present a technique for the specification and symbolic verification of graph-shaped systems, such as network protocols, based on graph transformation.

Our program model represents configurations using *hypergraphs*. A hypergraph is a graph with *hyperedges* — labelled edges of arbitrary finite arity. Actions are represented as *rewrite rules*, which describe how to rewrite a part of a hypergraph.

We use a symbolic representation of hypergraphs, called *patterns*. A pattern contains a *positive condition* to say that a subgraph should be contained in a configuration, as well as a set of *negative conditions*, to express that certain subgraphs should not be contained.

We present a technique for the automatic analysis of some safety properties in this setting — namely those safety properties for which

all *bad states* can be expressed with patterns. A symbolic backward analysis is conducted, which over-approximates the set of states backwards reachable from some violating state. If the analysis terminates, we can check if the initial state was reached. If not, then the property is true. If it was reached, then there is still a possibility that the initial states were only reached due to over-approximation, but are not reachable by an exact computation.

As a main example, we verify routing loop freedom of the ad hoc routing protocol DYMO, which is currently on the IETF standards track, to potentially become an Internet standard [CP08]. We also verify some other graph-shaped systems; e.g., a firewall example.

6. Related Work

This thesis covers approximately the following topics/keywords (grouped by Paper):

A, C parameterized systems, safety and liveness, symbolic model checking for infinite-state systems, regular model checking, acceleration, the automata-theoretic approach, specification logic, Büchi acceptance, translation into Büchi automata;

B liveness as fair termination, liveness using backwards reachability without computing transitive closures, incomplete analysis of liveness;

D routing protocols, DYMO, graph grammars, rewriting, symbolic backwards reachability, patterns, negative application conditions (NACs).

Generalizations of Regular Model Checking

There is a large body of work related to regular model checking, where some modeling restrictions have been lifted, thus obtaining a more general framework.

By not requiring a *length-preserving* transition relation, we can more naturally model e.g. dynamic process creation and deletion. We can model a non-length-preserving transition relation as length-preserving by using so-called *padding symbols* to represent a "pre-allocated buffer" of arbitrary but fixed size. This is sufficient for the analysis of safety, but not in general for liveness.

Transition relations which are not length-preserving have been considered in [DLS01, BLW05]. Dams et al. [DLS01] present a technique for computing the transitive closure of a not necessarily length-preserving transition relation.

One difficulty which arises in this setting is that loop-finding (a.k.a. *cycle detection*) for the analysis of liveness (see Section 4.3) is no longer complete; i.e., there may exist counterexamples which are not loop-shaped — e.g., because the size of the state vector grows unboundedly. The analysis of safety and liveness is described for non-length-preserving transition relations, and for infinite words, by Bouajjani et al. in [BLW05], where a simulation property replaces loop-finding.

Boigelot et al. and Bouajjani et al. consider configurations modeled as infinite words instead of finite words, used to represent, e.g., real numbers [BLW04, BLW05].

Context-free languages, rather than regular languages, were considered in [BH99, FP01, Lan04], motivated by the need for non-regular invariants.

We need not limit ourselves to words. Several works consider tree or graph shaped configurations. Works close to regular model checking consider, e.g., regular sets of trees [KMM$^+$01, AJMd02, BT02, LS02, ALdR06].

We can also relax the assumptions on the *atomicity* of the program statements, by for example evaluating global conditions in guards of actions "non-atomically", allowing other actions to take place while the condition is being evaluated. This has been examined for parameterized systems by Abdulla et al. for the analysis of safety properties [AHDR08].

Acceleration

In symbolic model checking [BCM$^+$90], we iteratively compute successors (or predecessors) with a transition relation until a fixed-point has been reached. For parameterized and infinite-state systems, such a computation need not terminate; in fact any non-trivial model checking problem is undecidable [AK86].

To make the fixed-point computation terminate more often, so-called *acceleration* techniques have been developed which compute the transitive closure α^+ of an action α as an aid to compute $Pre^\star(\longrightarrow, S)$ or $Post^\star(\longrightarrow, S)$ for some set of states S and a transition relation $\longrightarrow$ with $\alpha \subseteq \longrightarrow$.

Acceleration techniques have been developed for regular model checking [JN00, BJNT00, DLS01, AJNd02, BLW03]. Here, α^+ is not computable in general; however, computability results have been established for some classes of actions [JN00]. Finkel et al. developed systematic acceleration techniques for programs with a finite number of variables, typically ranging over integers [FL02, BFLS05].

There is much work on extending symbolic model checking to different classes of infinite-state systems. Acceleration techniques have been developed for systems with unbounded FIFO channels [BP96, BGWW97, ABJ98, BH99, ACABJ04], with stacks [Cau92, BEM97, Cau92, FWW97, ES01, EKS06], with counters [BW94, CJ98, BW02, BLW03], and for *Petri nets* [Mur89]. These works define a specific symbolic representation — e.g., *SRE* [ABJ98] for lossy FIFO channels, and *QDD* [BP96, BGWW97] for FIFO channels — for sets of states, and give specialized procedures for computing the set of reachable states. A scheme for extending symbolic model checking to infinite-state systems was presented by Kesten et al. in

[KMM+01], and illustrated for the analysis of safety. Usually the verification problem is undecidable, with exceptions for e.g. *pushdown systems* [BEM97, FWW97, ES01, EKS06] and *well-structured systems* [ACJT00].

A related approach is *widening*, or *extrapolation*, where a (sometimes exact) over-approximation of α^+ or $Post^\star(\alpha, S)$ for some set of states S is computed [BJNT00, Tou01, BLW03].

Abstraction

An *abstract* version of a program is a "simplification" of the program, with less "detail". As such, it over-approximates the behavior of the program. The point of abstraction is to simplify the verification, by obtaining a simpler program from which we can draw conclusions about the original system [CC77, Sif84, CGL92, LGS+95, KP00]. Many techniques based on *abstraction* compute an over-approximation of the set of reachable states (or, for the analysis of liveness, of the reachable loops). Any abstract system which over-approximates the set of reachable states of the original (*concrete*) program has the following property: if the abstract system satisfies a safety property, then so does the concrete one. The analogous holds for over-approximation of the fair computations, for liveness properties. Many different abstraction based techniques exist. A brief overview follows.

In [KP99, KPV01], Kesten et al. introduce so-called *verification by augmented finitary abstraction (VAA)*. This is a *finitary* abstraction framework — meaning that the obtained abstract program is finite-state — for infinite-state model checking of safety and liveness (LTL) which is sound and complete, in the sense that: (1) it abstracts an (infinite-state) Büchi automaton into a finite-state Büchi automaton; (2) if the abstract system satisfies the property, then so does the concrete one; and (3) whenever an infinite-state Büchi automaton has no computations (i.e., the system satisfies a property), there exists a finitary abstraction into a corresponding finite-state Büchi automaton which has no computations. The framework gives us another complete approach to verification. In order to effectively automatize the approach, heuristic support for finding a suitable ranking function and an abstraction function is needed.

Another interesting approach is the use of *counter abstraction* by Pnueli, Xu, and Zuck [PXZ02, ZP04], where a finite abstract system is constructed on the principle that there is one counter for each local state, which counts how many processes are in that local state: 0, 1, or ∞ (where ∞ means "> 1"). Guidelines for finding sound abstract fairness requirements were also given. A potential problem is the blow-up of the number of states of the finite-state system. For example, a parameterized system where each process has m possible local states results in an abstract finite-state system with up to 3^m states. For example,

Szymanski's algorithm [Szy90] can be modeled (see Paper B) for safety as having $7 \times 2 \times 2$ local states, resulting in a counter abstracted system with 3^{28} states (roughly 2×10^{13}). In [PXZ02], however, a reduced model is used, using only 7 counters. The local state has been reduced to essentially one variable, the program counter, which is motivated since the values of w and s are functions of the program counter. This gives a model of at most 3^7 (roughly 2000) states for safety, and up to around 15000 states for liveness. Establishing safety required under a second for this reduced model, and liveness under two minutes,

When using counter abstraction to analyze properties which distinguish individual processes, a modification is required. Consider for example an *individual* liveness property of form $\forall i \, \Box(S[i] \rightarrow \Diamond T[i])$ where i ranges over process indices (see Paper A). A counterexample to this property is that some distinguished process k at some point in time satisfies $S[k]$ but thereafter never $T[k]$. An abstract state then contains an exact representation of the local state of process k, while the other processes are counter abstracted as before.

A technique for the analysis of safety properties uses *upward closed sets* [ACJT00] as a symbolic representation of configurations. The representation is upward closed with respect to an ordering on the configurations, which must be provided. For example, for a system where a configuration is a graph, one could choose "subgraph" as the ordering, with the implication that a graph should simulate all its supergraphs (i.e. the graphs which it is a subgraph of). A limitation is that universal constraints (i.e., using a $\forall$ quantifier) cannot be represented exactly as upward closed sets, whereby an over-approximation is introduced. Upward closed sets have been used recently to verify parameterized systems [ADHR07, ADR07]. In some sense, Paper D is also based on upward closed sets, as the symbolic representation used — *patterns* — are upward closed sets.

When using over-approximation in abstractions, a *spurious* counterexample may be found by the model checker, i.e., a false alarm, which exists only in the abstract program. Then the abstraction should be *refined*, so that the same counterexample is not found again, and the analysis restarted.

A general approach, where the abstraction is automatically refined, is so-called *counterexample-guided abstraction refinement (CEGAR)*. The technique was introduced in [CGJ$^+$00] by Clarke et al. as a complete procedure for finite-state model checking (for a fragment of *CTL** without existential path quantification). Sufficient conditions for sound abstraction are given.

A CEGAR approach was presented for regular model checking by Bouajjani et al. in [BHV04]. Their approach for checking liveness is to first translate the system into a Büchi regular transition system

which accepts all counterexamples, as in the automata-theoretic approach [Var91]. Then the loop-finding is reduced to reachability (as in [SB04] — see "Checking Liveness as Safety" below), after which abstractions are applied in the computation of the reachable states. This results in an over-approximation of the counterexamples, and is therefore sound. Since the verification problem is undecidable, the refinement loop need not terminate. This approach is interesting, and proven effective at least for safety [BHV04], but it is difficult to foresee which abstraction schemes are suitable, in particular for the verification of liveness. To understand which abstraction schemes are efficient for liveness, I believe it would be useful to have specialized abstractions for fairness instead, as in [CGJ+00, PXZ02, ZP04].

Fair Termination and Liveness

An established approach for the analysis of safety and liveness is that of *deductive verification*, which requires to validate a list of premises of a proof rule, in order to establish a property of the system [MP91, MP95, MP96]. For liveness, we can use *ranking functions* to measure progress towards a goal, and *helpful directions*, which say which transition promotes progress towards the goal, to establish the validity of a property under (weak) fairness [LPS81, MP91, MP96]. Fang et al. [FPPZ06] develop rules for the verification of liveness of parameterized systems. Their approach is to automatically compute assertions needed to apply a proof rule, which they provide for liveness, and then use a small model theorem to reduce the problem to checking a small instance of the parameterized system.

Podelski and Rybalchenko [PR04, PR05] and Cook [CPR05] develop novel techniques for the verification of liveness leading up to the state-of-the-art liveness tool TERMINATOR [CPR06a, CPR06b].

In [PR04], Podelski and Rybalchenko give a theoretical result on how to prove liveness. They use an over-approximation of the transitive closure of the transition relation — a so-called *transition invariant*. They prove that a program $\mathcal{P}$ terminates if and only if there exists a *disjunctively well-founded* transition invariant for $\mathcal{P}$, i.e., a transition invariant which is contained in the finite union of a set of well-founded relations. Intuitively, this result reduces the termination proof to proving termination of smaller transition relations whose union contains the transition relation. To establish that a liveness property φ holds for a program $\mathcal{P}$ one has to validate that (1) T is a transition invariant for R, and (2) $T \cap (W_F \times W_F)$ is disjunctively well-founded; where R is the transition relation of a Büchi automaton $\mathcal{A}$ accepting the computations of $\mathcal{P}$ which violate φ, and W_F are the accepting states of $\mathcal{A}$ [Var91]. To establish (2) for $T = T_1 \cup \cdots \cup T_n$ we should prove that each "disjunct" T_i is well-founded, which is a simpler problem for a well-chosen T than proving

that R is well-founded. No clue is given in this work on how to find T automatically.

In [PR05], Podelski and Rybalchenko extend predicate abstraction to so-called *transition predicate abstraction*, focusing on the abstraction of transitions instead of states, for the analysis of liveness under fairness. The work can be viewed as a step towards automating the method of transition invariants. Their approach is to first transform a program $\mathcal{P}$ into a finite *abstract-transition* program $\mathcal{P}^{\#}$, which is a graph where nodes are labeled with abstract transitions and edges by concrete transitions, and then marking nodes as *terminating* and *fair* after suitable tests, so that the liveness property is true if each fair node is marked terminating. Intuitively, the abstract program represents all compositions of actions. The marking of nodes as terminating, amounts to checking well-foundedness of abstract transitions. In order to apply their technique, one needs to find suitable *transition predicates* upon which to abstract — a transition predicate is an over-approximation of a program action. The approach has been proven effective for programs with linear arithmetic, using linear arithmetic predicates, but is difficult to automate for general programs. The method is not complete, but in [CPR05] a *counterexample-guided abstraction refinement* algorithm is presented.

Checking Liveness as Safety

In [SB04] Schuppan and Biere present a reduction from liveness checking to safety checking for finite-state systems. Actually, the reduction finds any *loop-shaped* counterexample, and therefore it is "as complete" as the loop-finding procedure we described in Chapter 4.

As a brief example (for details, see [SB04]), we explain their reduction applied to the formula $\Diamond S$ ("eventually S" expressed in *LTL* — see [Pnu77] or Paper A). The reduction is based on extending the system with a variable *looped* which keeps track of whether a loop is closed (a loop can start non-deterministically at any time), and a variable *live* which checks whether S has been seen. The liveness formula is thus equivalent to the safety formula $\Box(\mathit{looped} \rightarrow \mathit{live})$. The loop is checked by a state-recording construction, where a copy of the state where the loop starts is stored, and the end is determined by comparison with the stored starting state. The start of the loop is non-deterministically chosen (hence it can start in any state).

In [SB06] Schuppan and Biere extend the reduction in a straightforward way to many classes of infinite-state systems, including regular transition systems (for which loop-finding is complete). It seems that this construction is equivalent with the way loops are checked in [Nil05] and Paper A (see also Section 4.3). We do not, however, explicitly encode the variables *looped* and *live* — it is implicit in the computation of

the transitive closure. Schuppan and Biere keep a copy of the configuration where the start of a loop is stored, and use the variables *looped* and *live*, as described above. The reduction to safety essentially computes the transitive closure of the transition relation, as the recorded state can be any reachable state and the loop can start at any state. One difference is that in our description at Section 4.3, the reachable states are first computed, before looking for loops, but this is not strictly necessary — we could directly compute the transitive closure (which however is more expensive for our examples).

Regular Inference

Habermehl and Vojnar use *regular inference*, a.k.a. *learning*, to verify safety properties of parameterized systems [HV05]. Their approach is based on the observation that there exist techniques for inferring a regular language based on positive and negative samples of accepted words [TB73, Ang87]. This can be used to verify regular transition systems, as follows. Suppose that the regular transition system represents a parameterized system $\mathcal{P}(n)$ where n is the number of processes. The goal is to check whether, for some n, the set of reachable configurations of $\mathcal{P}(n)$ can reach a set of bad configurations. The procedure either finds an inductive invariant strong enough to prove the property, or finds a counterexample, or it does not terminate. Termination is guaranteed if the set of reachable configurations is regular. The procedure works as follows. First compute the reachable configurations of the (finite) instance $\mathcal{P}(1)$ — call them Inv_1 — which are words of length 1 (optionally, we could start with a higher instance). Observe that Inv_1 should be accepted, and any other word of length 1 should not be. Continue in this way, by using $Inv_2, Inv_3, \cdots$ as samples of what should be in the inductive invariant and their (length-wise) complements as negative samples. If after some iteration, the inferred language is an inductive invariant (i.e., includes the initial configurations and is closed under the transition relation) and does not intersect the set of bad configurations, the property is true. Similarly, if the inferred language intersects the set of bad configurations, the property is false.

Reduction to Small Instances

A number of techniques are based on a proof that it is sufficient to check a small instance of a parameterized system, in order to guarantee that the property holds for all instances. The small instance can then be verified using finite-state techniques.

We have already mentioned the work by Fang et al. [FPPZ06]. The approach was also followed by Emerson, Namjoshi and Kahlon [EN95, EK00, EK02, EK04]. In [EK02] the reduction of the uniform verification problem to a finite-state verification problem for three or fewer processes

works for both safety and liveness. The systems considered are so-called *resource allocation systems* consisting of an arbitrary finite number of processes, where each process can acquire a set of tokens, and release all its tokens, or take a local transition. However, modeling of *global conditions* as guards for actions — such as "if all other processes are in a specific local state, then take a local transition" — is limited or impossible in this setting, but occurs in many parameterized protocols we have studied.

7. Conclusions and Future Work

We have developed and improved methods for the verification of correctness properties of parameterized and infinite-state systems. Most of our experiments have been made in the framework of regular model checking (in Papers A, B, C), motivated by the difficulty of verifying liveness. Now we summarize our experiences, draw conclusions, and elaborate on future work based on ours. Afterwards we "zoom out" and give our personal view on interesting directions for future work, also taking into account related work in the community.

Conclusions and Future Work by Paper

A. *Regular model checking for LTL(MSO).* We extended the automata-theoretic approach [Var91] to regular transition systems. The automatic translation is valuable as it allows us to quickly and correctly translate models into automata.

In order to obtain a stand-alone model checker for regular model checking, a number of practical (and hopefully theoretically challenging) tasks remain. One direction is to optimize the translation of the logic into automata, on which much effort has been spent in, for example, MONA [KMS02].

Natural directions are to extend the logic to other models for which related techniques exist, such as tree- or graph-shaped structures.

Our acceleration techniques work best when applied to individual actions. One issue with writing the system and negated property as one formula, and translating this formula into an automaton, is that we no longer recognize the individual actions. We do, however, want a uniform model checker, which searches for models of any formula. We conclude that the way to go to achieve both a uniform model checker and to conform with current verification techniques is to accelerate "parts" (subrelations) of the automaton obtained after translation, which correspond to the program actions.

Hence, we extend the objectives for acceleration techniques so that whenever we construct an acceleration scheme for a class of actions, we should also figure out how to efficiently "extract" such actions from an automaton representing the entire transition relation. This

approach was pursued for one class of actions, with good results, in Paper C.

B. *Proving liveness by backwards reachability.* We have developed a backward reachability based method for proving liveness, and given two techniques to prove liveness. One technique is commutativity based, and the other is based on "executing statements in some order".

Theoretical results which establish completeness for classes of programs, or relates the two techniques, are lacking. A second problem is that the "executing statements" technique does not say which action to choose, in a general setting. There is more development possible here, possibly leading up to a collection of reachability based proof techniques. We should consider many (commonly occurring) ways in which a set of states can terminate in ω steps.

Our experiments were for regular model checking. It would be interesting to see how our techniques perform in frameworks where backward reachability is extra efficient (even decidable). In order to improve the performance of these techniques, clearly, one should optimize the backward reachability computation itself. In the context of regular model checking, this can be done on a high level by optimizing the acceleration techniques (we know how to do this efficently for many classes of actions). On a lower level, one must look at the automata representation, and the minimization and determinization. Finally, heuristic support for choosing which technique to apply when is necessary for efficient automatization.

C. *Systematic acceleration in regular model checking.* We developed a technique for efficiently accelerating a class of actions, in a systematic and automatic way, using compositions of actions.

One conclusion is that automatic composition and acceleration gives us acceleration power "beyond modeling ingenuity". Concretely,

(a) compositions of actions which were before manually added to the model, necessary for verification, were now found automatically

(b) our technique gave us accelerations sufficiently strong to verify liveness of a program, but we could not reproduce the result by identifying which compositions were necessary and adding them to the model manually.

In other words, our experiments witness that we can not only reproduce "modeling ingenuity", we can also outperform it. Thus we have reduced the need for smart modeling somewhat.

We can apply these acceleration techniques on an automaton which represents the entire transition relation, by first extracting the actions from the automaton. Hence, these techniques can, in principle, be applied on any automaton output by the logic translation of Paper A.

An important result of this work is that using these techniques we can compute an exact representation of the reachable fair loops of our entire benchmark. An important conclusion is therefore that composition and acceleration is sufficient to prove safety and liveness (at least, many properties) for the protocols considered.

D. *Graph grammar modeling and verification of ad hoc routing protocols.* We developed a symbolic backward reachability framework for graph transformation systems, based on using so-called *patterns* as a symbolic representation. We use a subsumption ordering $\preceq$ on patterns, with the semantics that $\varphi \preceq \psi$ iff $[\![\varphi]\!] \subseteq [\![\psi]\!]$. The framework is "almost" that of well-structured systems [ACJT00] — except that the transition system used is not monotonic and not well quasi-ordered (!). The transition system is not monotonic, because of the negative conditions. For a better understanding, see the proofs in Paper D. A proof sketch for why it is not well quasi-ordered follows.

Consider a sequence of patterns $(\varphi_i)_{i \geq 1}$ where φ_i is of the form:

where we have used $\circ$ to represent absence of a node and $\bullet$ to represent existence (see Paper D for the exact syntax and semantics of patterns). This sequence forms an infinite anti-chain (i.e. for no $i < j$ do we have $\varphi_i \preceq \varphi_j$). Intuitively, φ_i represents the constraint "there exists a sequence of exactly i nodes, connected via edges labeled p".

Regardless, our main example is relatively large, and the number of patterns obtained in the backward analysis is so great that a theoretical guarantee for termination would not comfort much — rather, the focus has been on reducing the number of patterns which must be considered, using various optimizations.

Directions for future work include developing a counterexample-guided abstraction scheme for this framework. Practical tasks include optimizing the predecessor computations further.

Conclusions about Papers A, B and C

Papers A, B and C have the verification of liveness of parameterized systems in common. While Paper B introduces a general backward reachability based framework for proving liveness, we have implemented it for regular model checking.

The methods of Papers A and C are both complete, in the sense that (if the analysis terminates) they can answer whether a liveness property holds for the system. The method of Paper B on the other hand, is not complete, and therefore it is possible that the analysis terminates without answering our question.

In some sense, Paper C extends Paper A, as Paper C uses the same complete verification set up, but gives better verification results for our benchmark.

The motivation for Paper C originally came from Paper A. Our acceleration techniques work best when applied to individual program actions, but the formula obtained from the logic model, which describes the system conjoined with the negated correctness property, hides the actions. In Paper A we syntactically extracted them from the mentioned formula. In order to obtain syntax independence, Paper A pointed to a technique for semantically extracting actions. An alternative would be to keep the program actions separated during the translation process, but it is currently unclear how to achieve this. The technique of Paper C is useful independently of how the actions are represented — whether it is as one automaton representing their union, or as separate automata.

The performance improvement in Paper C relative to Paper A is essentially due to the improved acceleration power. By composing the actions before acceleration, we find sufficiently strong accelerations for our benchmark automatically, which would require much ingenuity from the modeller to find manually. The verification times in Paper C are slightly better than those in Paper A, and liveness properties of more programs are proven. The token pass and token ring examples were not included in Paper C as the new technique introduced is superfluous for them (since no compositions are necessary), and the focus in Paper C was on unary actions.

In comparison with the computation of the reachable loops, done in Papers A and C, the backward reachability based method of Paper B has to conduct many reachability computations, instead of one transitive closure computation. A comparison of the experimental results of Papers B and C show that no single approach is uniformly faster — it depends on the programs considered.

We conclude by emphasizing what is necessary to optimize the approaches. The main cause for slow performance in Paper B is backward reachability computations with large automata. Time spent on acceleration, whenever acceleration is used, can be more or less eliminated,

by using "pattern-based"/precomputed acceleration schemes, known for many types of actions [ABJN99]. To some extent, we can use optimizations to avoid unnecessary reachability computations, and we can try to keep the automata small. Once that has been done, further performance gains must come from lower level optimizations, such as focusing on representation, minimization and determinization of automata. Such lower level optimizations of course pay off for all regular model checking based verification, including Papers A and C.

Other Directions for Future Work

Let us now look at the area as a whole. Starting out on a high level, it would be desirable to set up a common specification language, or a minimal set of specification languages, for parameterized systems, and other programs of interest, and effectively set up a library of program models. This would encourage more competition, and a quicker development of the field. Annual competitions could then be held, much like the SAT-solver competitions we have seen lately.

Approximation based techniques have been proven successful in many problem domains. My opinion is that a CEGAR approach, for both safety and liveness, or just liveness, with specific handling of fairness, should be developed for regular model checking. We are not far from this, as a CEGAR approach for regular model checking has already been developed by Bouajjani, Habermehl and Vojnar [BHV04]. However, the approach chosen there was to verify liveness by applying abstraction after the automata-theoretic translation was applied, which perhaps hides the problems involving fairness. They did not report verification of the examples we have verified, which further motivates another go at abstractions for liveness. Other works handle fairness explicitly [KP00, CGJ+00, PXZ02]. As mentioned, another interesting approach is to continue the development of reachability based liveness techniques.

The learning based approach of Habermehl and Vojnar [HV05] is potentially useful in a wider context, such as acceleration and liveness, as well. A performance check would be interesting.

If we relax the assumptions on the atomicity of program statements, e.g., by allowing other actions to take place while a global condition is being evaluated, the linear structure of programs is lost, and regular model checking cannot be used. This has been examined for parameterized systems by Abdulla et al. for the analysis of safety properties [AHDR08]. Further development in this direction is possible.

Finally, there is today an abundance of techniques for regular model checking. Perhaps it is due time to write a unifying tool, with plug-in support for different methods, and a common specification language — suitably something like LTL(MSO) — with all the benefits that brings. This would promote further development of the field.

Bibliography

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 305–318. Springer-Verlag, 1998.

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 134–145. Springer-Verlag, 1999.

[ACABJ04] Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. Using forward reachability analysis for verification of lossy channel systems. *Formal Methods in System Design*, 25(1):39–65, 2004.

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 313. IEEE Computer Society, 1996.

[ACJT00] Parosh Aziz Abdulla, Kārlis Cerāns, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1–2):109–127, 2000.

[ADHR07] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS '07: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 721–736, 2007.

[ADR07] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV '07: Proceedings*

of the 19th International Conference on Computer Aided Verification, pages 145–157, 2007.

[AHDR08] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In *VMCAI '08: Proceedings of the 9th International Conference on Verification, Model Checking and Abstract Interpretation*, 2008. To appear.

[AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 555–568. Springer-Verlag, 2002.

[AJN+04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. Regular model checking for LTL(MSO). In R. Alur and D. Peled, editors, *Proceedings 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 348–360. Springer Verlag, 2004.

[AJNd02] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*, pages 116–130. Springer-Verlag, 2002.

[AJNd03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Algorithmic improvements in regular model checking. In *CAV '03: Proceedings of the 15th International Conference on Computer Aided Verification*, pages 236–248, 2003.

[AJNS04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A survey of regular model checking. In P. Gardner and N. Yoshida, editors, *Proceedings CONCUR 2004, 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer Verlag, 2004.

[AJRS06] Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena. Proving liveness by backwards reachability. In C. Baier and H. Hermanns, editors, *Proceedings*

CONCUR 2006, 17th International Conference on Concurrency Theory, volume 4137 of *Lecture Notes in Computer Science*, pages 95–109. Springer Verlag, 2006.

[AK86] Krzysztof Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

[ALdR06] Parosh Aziz Abdulla, Axel Legay, Julien d'Orso, and Ahmed Rezine. Tree regular model checking: A simulation-based approach. *Journal of Logic and Algebraic Programming*, 69(1–2):93–121, 2006.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer-Verlag, 1999.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[BFLS05] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *Proceedings of ATVA 2005, the 3rd International Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.

[BGWW97] Bernard Boigelot, Patrice Godefroid, Bernard Willems, and Pierre Wolper. The power of QDDs (extended abstract). In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 172–186. Springer-Verlag, 1997.

[BH99] Ahmed Bouajjani and Peter Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1–2):211–250, 1999.

[BHJM05] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5–6):505–525, 2007. Invited to special issue of selected papers from FASE 2004/05.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In *CAV '04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 372–386, 2004.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 403–418. Springer-Verlag, 2000.

[BLARS07] Igor Bogudlov, Tal Lev-Ami, Thomas W. Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *CAV '07: Proceedings of 19th International Conference on Computer Aided Verification*, pages 221–225, 2007.

[BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. In *Proceedings 15th International Conference on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235. Springer-Verlag, July 2003.

[BLW04] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega-regular model checking. In *Proceedings 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, pages 561–575, 2004.

[BLW05] A. Bouajjani, A. Legay, and P. Wolper. Handling liveness properties in (ω-)regular model checking. In *Proceedings of the 6th International Workshop on Verification of Infinite-State Systems (Infinity'04)*, volume 138 of *Electronic Notes in Computer Science*, pages 101–115, 2005.

[BP96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 1–12. Springer Verlag, 1996.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3. ACM, 2002.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BT02] Ahmed Bouajjani and Tayssir Touili. Extrapolating tree transformations. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 539–554. Springer-Verlag, 2002.

[Bur81] James Edward Burns. *Complexity of communication among asynchronous parallel processes*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1981.

[BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 55–67. Springer-Verlag, 1994.

[BW02] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: An overview. In *Proceedings of the 18th International Conference on Logic Programming (ICLP 2002)*, pages 1–19, 2002.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.

[Cau92] Didier Caucal. On the regular structure of prefix rewriting. In *CAAP '90: Selected papers of the conference on Fifteenth colloquium on trees in algebra and programming*, pages 61–86. Elsevier Science Publishers Ltd., 1992.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CCF+07] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée, invited paper. In *Proceedings of the first IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE '07*, pages 3–17. IEEE Computer Society Press, 2007.

[CCG+04] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, 2004.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1981.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169. Springer-Verlag, 2000.

[CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343–354, New York, NY, USA, 1992. ACM.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CH85] Thierry Coquand and Gérard P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL '85: Invited Lectures from the European Conference on Computer Algebra — Volume 1*, pages 151–184. Springer-Verlag, 1985.

[CJ98] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 268–279. Springer-Verlag, 1998.

[CM97] Stephen A. Cook and David G Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications*, volume 35, pages 1–17. American Mathematical Society, 1997.

[Coe95] T. Coe. Inside the Pentium FDIV Bug. *Dr. Dobbs Journal*, 20(4):129–135, 1995.

[CP07] Ian D. Chakeres and Charles E. Perkins. Dynamic MANET on-demand routing protocol. *IETF Internet Draft*, 2007. draft-ietf-manet-dymo-10.txt, work in progress.

[CP08] Ian D. Chakeres and Charles E. Perkins. DYMO — Dynamic MANET On-demand Routing Protocol home page. `http://www.ianchak.com/dymo/`, 2008.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *Proceedings of the 12th International Symposium on Static Analysis (SAS 2005)*, pages 87–101, 2005.

[CPR06a] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 415–426. ACM, 2006.

[CPR06b] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV '06: Proceedings of the 18th International Conference on Computer Aided Verification*, pages 415–418, 2006.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[DLS01] Dennis Dams, Yassine Lakhnech, and Martin Steffen. Iterating transducers. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 286–297. Springer-Verlag, 2001.

[Dow97] Mark Dowson. The Ariane 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[EK00] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 236–254. Springer-Verlag, 2000.

[EK02] E. Allen Emerson and Vineet Kahlon. Model checking large-scale and parameterized resource allocation systems. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 251–265. Springer-Verlag, 2002.

[EK04] E. Allen Emerson and Vineet Kahlon. Parameterized model checking of ring-based message passing systems. In *Pro-*

ceedings of the 18th International Workshop on Computer Science Logic (CSL), pages 325–339, 2004.

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 489–503, 2006.

[EN95] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–94. ACM, 1995.

[EN96] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 87–98. Springer-Verlag, 1996.

[ES01] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 324–336. Springer-Verlag, 2001.

[FL02] S. Finkel and J. Leroux. How to compose presburger-accelerations: Applications to broadcast protocols. In *FSTTCS '02: Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. In *FSTTCS '01: Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 156–170. Springer-Verlag, 2001.

[FPPZ06] Yi Fang, Nir Piterman, Amir Pnueli, and Lenore Zuck. Liveness with invisible ranking. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):261–279, 2006.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Proceedings 2nd International Workshop on Verification of In-*

finite State Systems (INFINITY'97), volume 9 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1997.

[Gat95] Bill Gates. Interview. *Focus: das moderne Nachrichtenmagazin*, 43:206–212, 1995.

[GMW79] Michael Gordon, Arthur Milner, and Christopher Wadsworth. Edingburgh LCF: A mechanised logic of computation. *Lecture Notes in Computer Science*, 78, 1979.

[Gon04] G. Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge, 2004.

[GS92] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[Hal00] Thomas C. Hales. Cannonballs and honeycombs. *Notices of American Mathematical Society*, 47(4):440–449, 2000.

[HDPR02] John Hatcliff, Matthew B. Dwyer, Corina S. Păsăreanu, and Robby. *Foundations of the Bandera abstraction tools*, pages 172–203. Springer-Verlag New York, Inc., 2002.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Automata Theory, Languages, and Computation, 3rd Edition*. Addison-Wesley, 2007.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, 1997.

[HV05] Peter Habermehl and Tomás Vojnar. Regular model checking using inference of regular languages. *Electronic Notes in Theoretical Computer Science*, 138(3):21–36, 2005.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 220–234. Springer-Verlag, 2000.

[Joh78] S. C. Johnson. *Lint, a C Program Checker*. AT & T Bell Laboratories, 1978.

[JS07] Bengt Jonsson and Mayank Saksena. Systematic acceleration in regular model checking. In W. Damm and H. Hermanns, editors, *Proceedings 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 131–144. Springer Verlag, 2007.

[Kin94] E. Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

[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):93–112, 2001.

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.

[KP99] Yonit Kesten and Amir Pnueli. Verifying liveness by augmented abstraction. In *CSL '99: Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic*, pages 141–156. Springer-Verlag, 1999.

[KP00] Yonit Kesten and Amir Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000.

[KPV01] Yonit Kesten, Amir Pnueli, and Moshe Y. Vardi. Verification by augmented abstraction: the automata—theoretic view. *Journal of Computer and System Sciences*, 62(4):668–690, 2001.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lam79] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):84–97, 1979.

[Lam86] Leslie Lamport. The mutual exclusion problem: parts I and II. *Journal of the ACM (JACM)*, 33(2):313–348, 1986.

[Lan04] Martin Lange. Symbolic model checking of non-regular properties. In *CAV '04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 83–95, 2004.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LPS81] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 264–277. Springer-Verlag, 1981.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):134–152, 1997.

[LS02] D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 274(1–2):89–115, 2002.

[LT93] Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[Mai01] Monika Maidl. A unifying model checking approach for safety properties of parameterized systems. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 311–323, 2001.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MP84] Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, 1984.

[MP91] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., 1995.

[MP96] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Progress*. Draft, 1996.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[Mye79] G. J. Myers. *The Art of Model Checking*. John Wiley & Sons, 1979.

[NBS06] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame graphs. In *Automated Reasoning (IJCAR 2006)*, pages 21–35. Springer, 2006.

[Nil05] Marcus Nilsson. *Regular Model Checking*. PhD thesis, Uppsala University, 2005.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In *Applied Formal Methods — FM-Trends 98*, pages 338–345. Springer-Verlag, 1998.

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[Pnu77] Amir Pnueli. The temporal logic of programs. *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (TSE)*, pages 46–57, 1977.

[PR04] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS '04: Proceedings of the 19th Annual IEEE*

Symposium on Logic in Computer Science (LICS'04), pages 32–41. IEEE Computer Society, 2004.

[PR05] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 132–144. ACM, 2005.

[PS00] Amir Pnueli and Elad Shahar. Liveness and acceleration in parameterized verification. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 328–343. Springer-Verlag, 2000.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 107–122. Springer-Verlag, 2002.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.

[SB04] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2):185–204, 2004.

[SB06] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electronic Notes in Theoretical Computer Science*, 149(1):79–96, 2006.

[Sif84] Joseph Sifakis. Property preserving homomorphisms of transition systems. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 458–473, London, UK, 1984. Springer-Verlag.

[Sis85] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 39–48. ACM, 1985.

[SSE05] Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. jMoped: A Java bytecode checker based on moped. In *TACAS '05: Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 541–545, 2005.

[SWJ08] Mayank Saksena, Oskar Wibling, and Bengt Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*. Springer Verlag, 2008. To appear.

[Szy90] B. K. Szymanski. Mutual exclusion revisited. In *JCIT: Proceedings of the fifth Jerusalem conference on Information technology*, pages 110–119. IEEE Computer Society Press, 1990.

[TB73] B. Trakhtenbrot and J. Barzdin. *Finite Automata: Behavior and Synthesis*. North-Holland, 1973.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 133–191. MIT Press, 1990.

[Tou01] Tayssir Touili. Regular model checking using widening techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Workshop on Verification of Parameterized Systems (VEPAS'01).

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936.

[Var91] Moshe Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1–2):79–98, 1991.

[Var07] Moshe Y. Vardi. Automata-theoretic model checking revisited. In *VMCAI '07: Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 137–150. Springer, 2007.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 88–97. Springer-Verlag, 1998.

[ZP04] Lenore D. Zuck and Amir Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3–4):139–169, 2004.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 419

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-8605

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2008

Paper A

Regular Model Checking for $LTL(MSO)$ (Extended Version)

The original version was published in LNCS, volume 3114.

Regular Model Checking for $LTL(MSO)$ (Extended Version)

Parosh Aziz Abdulla
parosh@it.uu.se
Uppsala University

Bengt Jonsson
bengt@it.uu.se
Uppsala University

Marcus Nilsson
marcusn@it.uu.se
Uppsala University

Julien d'Orso
juldor@it.uu.se
Uppsala University

Mayank Saksena
mayanks@it.uu.se
Uppsala University

Abstract

Regular model checking is a form of symbolic model checking for parameterized and infinite-state systems whose states can be represented as words of arbitrary length over a finite alphabet, in which regular sets of words are used to represent sets of states. We present $LTL(MSO)$, a combination of the logics MSO and LTL as a natural logic for expressing temporal properties to be verified in regular model checking. In other words, $LTL(MSO)$ is a natural specification language for both the system and the property under consideration. $LTL(MSO)$ is a two-dimensional modal logic, where MSO is used for specifying properties of system states and transitions, and LTL is used for specifying temporal properties. In addition, the first-order quantification in MSO can be used to express properties parameterized on a position or process. We give a technique for model checking $LTL(MSO)$, which is adapted from the automata-theoretic approach: a formula is translated to a *Büchi regular transition system* with a regular set of accepting states, and regular model checking techniques are used to search for models. We have implemented the technique, and show its application to a number of parameterized algorithms from the literature.

A.1 Introduction

Regular model checking is a framework for algorithmic symbolic verification of parameterized and infinite-state systems [BLW03, KMM+01,

WB98, BJNT00]. It considers systems whose states can be represented as finite words of arbitrary length over a finite alphabet, including array or ring-formed parameterized systems with an arbitrary number of finite-state processes, and systems that operate on queues, stacks, integers, and other linear unbounded data structures. In a system description, the set of initial states is represented as a regular set of strings, and the transition relation is given as a finite regular length-preserving transducer. Previous work on regular model checking [JN00, BJNT00, AJNd02] has developed methods for computing the set of reachable states of a system description, as well as the set of reachable loops, obtained from the transitive closure of the transition relation. In general, this problem is undecidable, but decidability results for certain classes have been obtained [JN00].

The techniques for computing reachable states and reachable loops can in principle be used to verify both safety and liveness properties of parameterized system descriptions, but do not provide a convenient approach for checking arbitrary temporal logic properties of parameterized and infinite-state systems. Significant ingenuity is required in order to manually transform the verification of a temporal property of a parameterized system into a property of reachable states and reachable loops, in particular if the verification uses fairness properties that are parameterized on system components [BJNT00, PS00]. It would be desirable to have a framework, analogous to the automata-theoretic approach in finite-state model checking [VW86], where the property of verifying a temporal property is automatically transformed into a problem of checking emptiness for a Büchi automaton.

In this paper, we address this problem by presenting an extension of the automata-theoretic approach [VW86] to the setting of regular model checking. We present a logic for expressing system models and temporal properties, which is a combination of the logics MSO over finite words and LTL. We use MSO for specifying sets of states and transition relations and LTL for specifying temporal constraints. The result is a two-dimensional modal logic, where MSO is used in the "space" (system state) dimension and LTL is used in the "time" dimension. Models of the logic are infinite sequences of (constant-length) words, representing computations of the specified system. We can then specify a verification problem as the conjunction of a system specification and a negation of the property to be verified.

Following the automata-theoretic approach, we present an automated translation from the translation from a formula φ in $LTL(MSO)$ to a *Büchi regular transition system (BRTS)*, consisting of a regular set I of initial states, a regular length-preserving transducer T, and a regular set F of accepting states. Accepting runs of the BRTS are infinite sequences of words, where the first word is in I, consecutive words satisfy T, and

infinitely many words are in F. We prove that φ is satisfiable if and only if the BRTS has an accepting run. Since T is length-preserving, the existence of an accepting run can be checked by searching for a reachable loop which contains a state in F.

A nice feature of our combination of MSO with LTL is that we get the power to express temporal properties parameterized over positions for free: MSO offers variables to represent positions and quantify over them, which can be interleaved with temporal operators. As a concrete example, for a parameterized mutual exclusion algorithm, a typical property one would want to express is the following.

> If all processes satisfy a weak fairness requirement, then each process that is interested in entering its critical section will eventually do so.

If the number of processes is fixed, the terms like "each process" can be replaced by explicit conjunctions to obtain a standard model checking problem in propositional temporal logic. However, for parameterized systems the number of processes is arbitrary. Fortunately, we can express this property directly in our logic, by a formula like

$$\forall i : \Box\Diamond[\text{blocked}(i) \vee \text{progressing}(i)] \longrightarrow \forall i : \Box\,[\text{trying}(i) \rightarrow \Diamond\text{critical}(i)]$$

where i ranges over positions in the state, and each position represents a process. In this formula, we apply LTL operators ($\Box$ and $\Diamond$) to formulas with the MSO variable i, and later use MSO quantification over i to express parameterized properties. In our logic $LTL(MSO)$, temporal operators can be applied to formulas with at most one free first-order variable and no free second-order variables. This restriction allows to express parameterized temporal properties (e.g., fairness constraints) of individual processes in a parameterized system, as well as temporal properties of pairs of adjacent processes (in positions i and $i+1$ using one free variable i). The restriction has to do with the translation into automata, explained in Section A.7.

A further nice property of adapting the automata-theoretic approach is that our transformation results in a uniform problem of checking for accepting runs, for which we can develop techniques that are more uniform than those presented in previous work [JN00, BJNT00, AJNd02]. We have extended our tool for regular model checking [AJNd03] to check whether BRTS have accepting runs. This is done in two steps. First, the set of reachable states are computed as $Inv = I \circ T^*$. Secondly, loops are found by identifying identical pairs in $(F \cap T \cap Inv) \circ T^*$. This computation is more uniform and more efficient than the approach to verification of temporal logic properties outlined in [BJNT00], which builds on computation of the transitive closure T^+ of the transition relation. We have verified safety properties with the tool for many of the

examples in our previous work, as well as liveness properties for some of the examples.

As special cases, when the formula contains no temporal operators, our method specializes into a decision procedure for MSO similar to that of MONA [HJJ+96], and when the formula contains no quantifiers our method specializes to ordinary (i.e. finite-state) LTL model checking.

The remainder of the paper is structured as follows. In the next two sections, we present the logic $LTL(MSO)$. Section A.4 illustrates how it can be used to model and specify parameterized algorithms. The model checking technique, including the translation to BRTS is presented and proven correct in Section A.7. Verification is discussed in Section A.8.

Related Work Kesten et al. [KMM+97] and Pnueli and Shahar [PS00] use the logic FS1S which has the expressive power of regular expressions, to specify sets of states of parameterized systems, just as we do with our logic. The difference is essentially that we have a higher level approach, considering all of (future) LTL [Pnu77], and automatic translation. However, unlike us, Kesten et al. [KMM+97] also consider a logic for trees.

Bouajjani, Legay, and Wolper [BLW05] independently (from us) characterize *global* and *local-oriented* properties in the framework of (ω-) regular model checking, and work out how to analyze such properties. They also consider ω-regular systems, i.e., systems where configurations are infinite words. However, they do not provide an automatic translation from a system and property description into a verification problem, as we do here.

Our logic $LTL(MSO)$ applied to words is related to existential monadic second-order logic ($EMSO$) on grids to define picture languages accepted by *tiling systems* (see e.g. [GR97]). Indeed, transducers over words can be considered as tiling systems where each transition represents a *tile*. Thus, it is expected that our logic $LTL(MSO)$ is equivalent to $EMSO$ on grids. However, the two logics come from different motivations. While $EMSO$ on grids is used to reason about *pictures*, our logic is used to reason about *parameterized structures over time*. When applied to the word structure, the two logics coincide.

In addition to the work on regular model checking, cited earlier, there is a large body of research on the problem of model checking parameterized systems of *identical* processes, in which there is no ordering between processes, and hence the system state can be represented as a multiset of process states (e.g., [BLS01, Del00, EK03, EK00, GS92]). This problem is substantially simpler, since ordering between processes need not be considered.

Emerson and Namjoshi [EN95] give a technique for verifying a restricted class of parameterized token-passing algorithms by reducing an arbitrary ring to a small fixed-size ring under certain conditions. These restrictions are substantially stronger than in our framework. Sistla [Sis97] uses Büchi automata over two dimensional languages (reminding of transducers) to specify network invariants when verifying systems by induction over their linear process structure. It is unclear what class of systems can be handled automatically by this technique.

The problem of checking liveness properties of array-shaped parameterized systems was considered by Pnueli and Shahar [PS00], who presented a technique for computing the transitive closure of a restricted class of transition relations. They also first manually employ abstractions to make the implementation terminate.

Pnueli, Xu, and Zuck [PXZ02] present an interesting use of specialized abstractions in order to prove absence of starvation properties for Szymanski's algorithm and the Bakery algorithm. The abstractions keep track of the number of processes with certain properties, and generate a finite-state system, which can be model-checked. The presented abstraction is specialized to prove non-starvation, and loses much information so that, e.g., safety properties can no longer be checked.

A.2 Introduction to $LTL(MSO)$

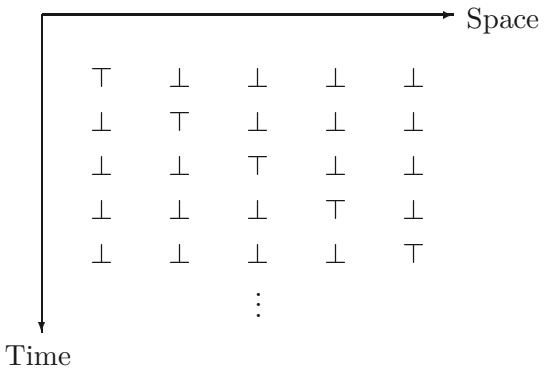
We introduce the logic $LTL(MSO)$ [AJN+04], intended for reasoning about infinite sequences of words of arbitrary length. Such sequences are useful to model executions of parameterized systems, where there are an arbitrary number of processes organized in a linear network. Each word in an execution models a system configuration, where each position in the word contains the local state of each process.

We follow the approach of *the Temporal Logic of Actions* by Lamport [Lam94], where both the protocol and the properties are specified by formulas in a single logic. Correctness of the protocol means that the formula specifying the protocol implies the formula specifying the property. We show how to specify protocols and properties using this logic and how to set up verification problems. Formulas in this logic can then be translated into BRTS, introduced in Section A.8, which can be used to find models of the original formula.

As a running example, we use a *token passing* protocol. It consists of an arbitrary number of processes organized in a linear array and numbered from 0 to $n-1$. The processes are ordered from left to right such that process 0 is the leftmost process and process $n-1$ is the rightmost process. Initially, the leftmost process has the token. In each step, a process can pass the token to its right neighbor. We model each

configuration as a word w over the alphabet $\{\bot, \top\}$ where the local configuration of process i is modeled by the symbol $w(i)$, i.e., the symbol at position i of the word. The symbol $\bot$ denotes a process that does not have the token, while the symbol $\top$ denotes a process that has the token.

In a system where configurations are modeled as words, an execution is an infinite sequence of words. All words in an execution have the same arbitrary length. Thus, we are working with two different dimensions. One dimension refers to the positions of the word, called the *space dimension*, and the other dimension refers to the points in time, called the *time dimension*. An execution of the token passing protocol is shown below; it can be seen as a *matrix* in which each element is indexed by a *timepoint* and a *position*, where the position refers to a process.



Formulas in $LTL(MSO)$ will be interpreted over such matrices. The logic consists of constructs for handling both the space and the time dimensions. Below, we introduce the constructs of $LTL(MSO)$ and illustrate with the token passing protocol.

Configuration Variables and Positions The atomic formulas are of the form $x[i]$ where x is a *configuration variable* and i is a *position variable*. The configuration variables model the global state of the protocol we are modeling. Each configuration variable contains a boolean variable for each position in the word, and is therefore essentially a boolean array (bit vector). The formula $x[i]$ denotes the boolean value of x at position i, at the timepoint at which the formula is interpreted. In the case of the token passing example, we use a configuration variable t such that $t[i]$ is true if and only if process i has the token.

MSO To specify configurations, i.e., the space dimension, we use *Monadic Second-Order Logic* (MSO) over words [Tho90, HJJ$^+$95], a logic that can express regular sets of words. It contains first-order position variables $i, j, \cdots$ denoting positions, and second-order position variables $I, J, \cdots$ denoting sets of positions. The atomic formulas of

MSO are of the form $i = j + 1$ (successor), $i \in I$, and $I \subseteq J$, where i, j are position variables and I, J are sets of position variables. A configuration variable x can be seen as a special case of a second-order variable, where $x[i]$ means $i \in x$, except that a configuration variable may change over time. Configuration variables are used for the purpose of modeling configurations, and always occur free in formulas. First-order quantification over positions and second-order quantification over sets of positions are allowed, for example the formula

$$\forall i : x[i]$$

can be used to specify that the configuration variable x is true at all positions. Using a combination of successor and quantification, we can express ordering, e.g., $\neg\exists j : i = j + 1$ can be used to express that $i = 0$. We can also express constant distances between positions of the form $i = j + c$ for any constant c, as well as the ordering $<$, using second-order quantification. We will use formulas with position variables like $x[i + 1]$ to mean $\exists j : j = i + 1 \wedge x[j]$.

In the token passing protocol, we can specify the initial condition that the first process has the token by the formula

$$\forall i : t[i] \longleftrightarrow i = 0$$

Primed Variables To specify transition relations, we need a relation between the current and the next timepoint. We use *primed* configuration variables for this, where $x'[i]$ is the value of x at position i at the next timepoint. In the token passing protocol, the transition relation where a process passes the token to its right neighbor is specified by

$$\exists i : \left[\begin{array}{ll} & t[i] \wedge \neg t'[i] \wedge \neg t[i+1] \wedge t'[i+1] \\ \wedge & \forall j \notin \{i, i+1\} : t'[j] = t[j] \end{array} \right]$$

Temporal Operators While MSO is used to reason about the space dimension, *linear temporal logic* (LTL) [Pnu77, Pnu82, MP92] is used to reason about the time dimension. The linear temporal logic adds the connectives $\Box$ *(always in the future)*, $\Diamond$ *(eventually)* and $\mathcal{W}$ (*weak until)*. In the token passing protocol, the following formula can be used to express that eventually the rightmost process has the token.

$$\Diamond\, \exists i : t[i] \wedge i = \$$$

where $i = \$$ means that i is the rightmost process (which can be expressed in MSO). Similarly, we can use the following formula to denote that there is always at least one token in the system

$$\Box\, \exists i : t[i]$$

Combining the two logics LTL and MSO, we obtain the logic $LTL(MSO)$ by allowing the position quantifiers and the temporal connectives to interleave. For example, we can express that at some point in time there is a process which from then on always has the token:

$$\Diamond\, \exists i : \Box\, t[i]$$

Given a formula φ representing a transition relation, we can use the formula $\Box\varphi$ to specify that all pairs of consecutive (in time) configurations will satisfy the constraints of the transition relation. The token passing protocol can thus be specified by conjoining the specification of the set of initial configurations and the transition relation:

$$\begin{array}{lll} & \forall i : t[i] \longleftrightarrow i = 0 & \\ \wedge & \Box\, \exists i : \left[\begin{array}{ll} & t[i] \ \wedge\ \neg t'[i] \ \wedge\ \neg t[i+1] \ \wedge\ t'[i+1] \\ \wedge & \forall j \notin \{i, i+1\} : t'[j] = t[j] \end{array} \right] \end{array}$$

Interleaving of position quantifiers and temporal operators will be restricted so that there can be at most one free position quantifier inside temporal operators (otherwise they cannot be translated — see Section A.7). For example,

$$\forall i : \Diamond\, \forall j : x[i] = y[j]$$

is allowed but not

$$\forall i : \forall j : \Diamond\, x[i] = y[j]$$

A.3 $LTL(MSO)$

We give the syntax and semantics of $LTL(MSO)$.

Syntax We assume a set of *configuration variables* $\mathcal{V}$, denoted by $x, y, z, \cdots$, a set of *first-order position variables*, denoted by $i, j, k, \cdots$, and a set of *second-order position variables*, denoted by $I, J, K, \cdots$. An $LTL(MSO)$ formula is inductively defined as follows.

$i \in I \mid I \subseteq J \mid i = j + 1$	Atomic MSO formulas
$x[i], x'[i]$	Configuration Variables
$\mathit{true} \mid \mathit{false}$	Boolean constants
$\varphi \ \vee\ \psi \mid \varphi \ \wedge\ \psi \mid \neg\varphi$	Propositional connectives
$\forall i : \varphi \mid \forall I : \varphi \mid \exists i : \varphi \mid \exists I : \varphi$	MSO Quantification
$\Box\, \varphi \mid \Diamond\, \varphi \mid \varphi\, \mathcal{W}\, \psi$	Temporal operators

We impose the restriction that in each subformula of the form $\Box\,\varphi$ or $\Diamond\,\varphi$ or $\varphi\,\mathcal{W}\,\psi$ there is at most one free first-order position variable and no free second-order position variable. For reference, let us simply call a formula with this restriction a *restricted formula*. The restriction is required for the translation of a formula into Büchi Normal Form, given later in Section A.7. It is well-known that the temporal operators $\mathcal{U}$ (*until*) and $\mathcal{R}$ (*release*) can be expressed using the operators above [MP92]. Hence we include all of (future) *LTL* [Pnu77]. We will use the following abbreviations.

$$
\begin{array}{lcl}
\varphi \longrightarrow \psi & \stackrel{\triangle}{=} & \neg\varphi \vee \psi \\
\varphi \longleftrightarrow \psi & \stackrel{\triangle}{=} & (\varphi \longrightarrow \psi) \wedge (\psi \longrightarrow \varphi) \\
\varphi(f(i)) & \stackrel{\triangle}{=} & \exists j : j = f(i) \wedge \varphi(j) \\
 & & \quad \text{where } f(i) \text{ is an expression over } i, \text{ such as } i+1 \\
i < j & \stackrel{\triangle}{=} & \forall K : \left[\begin{array}{ll} & i+1 \in K \\ \wedge & \forall k : [k \in K \longrightarrow k+1 \in K] \longrightarrow j \in K \end{array} \right] \\
i = 0 & \stackrel{\triangle}{=} & \neg\exists j : i = j+1 \\
i = \$ & \stackrel{\triangle}{=} & \neg\exists j : j = i+1
\end{array}
$$

Semantics *LTL*(*MSO*) formulas are interpreted over *matrices* M over $2^{\mathcal{V}}$ of dimension $\infty \times n$, for some $n > 0$, given as a parameter. We call the vertical (first) dimension *time*, and the horizontal (second) dimension *space*.

Let $\mathbf{N}$ be the set of natural numbers, and $\mathbf{Z}_n = \{0, \ldots, n-1\}$. The element $M(t,i) \subseteq \mathcal{V}$ for $t \in \mathbf{N}$ and $i \in \mathbf{Z}_n$ represents the system configuration at time t of position (or subsystem) i, which assigns truth values to the configuration variables $\mathcal{V}$ — the variables assigned true are included in $M(t,i)$, those assigned false are not. We denote by $M(t)$ the row $M(t,0)\ M(t,1) \cdots M(t,n-1)$. The row $M(t)$ represents the system configuration at time t.

In general, a formula φ depends on its free first- and second-order variables and a timepoint, and the configuration variables of M. A *valuation Val* is a mapping from first-order variables to $\mathbf{Z}_n$ and second-order variables to $2^{\mathbf{Z}_n}$. We define satisfaction of formulas, $(M, Val, t) \models \varphi$, with respect to a matrix M, a valuation *Val*, and a timepoint t as shown in Figure A.1. For a closed formula φ we denote by $M \models \varphi$ that $(M, \emptyset, 0) \models \varphi$.

$(M, Val, t) \not\models$	$false$	
$(M, Val, t) \models$	$true$	
	$i \in I$	if $Val(i) \in Val(I)$
	$I \subseteq J$	if $Val(I) \subseteq Val(J)$
	$i = j + 1$	if $Val(i) = Val(j) + 1$
	$x[i]$	if $x \in M(t, Val(i))$
	$x'[i]$	if $x \in M(t + 1, Val(i))$
	$\varphi \vee \psi$	if $(M, Val, t) \models \varphi$ or $(M, Val, t) \models \psi$
	$\varphi \wedge \psi$	if $(M, Val, t) \models \varphi$ and $(M, Val, t) \models \psi$
	$\neg\varphi$	if $(M, Val, t) \not\models \varphi$
	$\forall i : \varphi$	if for all $m \in \mathbf{Z}_n$ we have $(M, Val[i \mapsto m], t) \models \varphi$
	$\forall I : \varphi$	if for all $S \subseteq \mathbf{Z}_n$ we have $(M, Val[I \mapsto S], t) \models \varphi$
	$\exists i : \varphi$	if there exists $m \in \mathbf{Z}_n$ such that $(M, Val[i \mapsto m], t) \models \varphi$
	$\exists I : \varphi$	if there exists $S \subseteq \mathbf{Z}_n$ such that $(M, Val[I \mapsto S], t) \models \varphi$
	$\Box\,\varphi$	if for all $t' \geq t$ we have $(M, Val, t') \models \varphi$
	$\Diamond\,\varphi$	if there exists $t' \geq t$ such that $(M, Val, t') \models \varphi$
	$\varphi\,\mathcal{W}\,\psi$	if $(M, Val, t) \models \Box\,\varphi$ or there exists $t' \geq t$ such that $(M, Val, t') \models \psi$ and for all t'' with $t \leq t'' < t'$ we have $(M, Val, t'') \models \varphi$

Figure A.1: Semantics of $LTL(MSO)$. The valuation $Val[i \mapsto m]$ acts as Val except that it maps i to m. The valuation $Val[I \mapsto S]$ is defined analogously.

A.4 Modeling in $LTL(MSO)$

In this section, we discuss how to model systems and set up verifications problem in $LTL(MSO)$.

A.4.1 Specifying Systems

A *state formula* is a formula without temporal operators and primed variables, used for specifying constraints on only one configuration. An *action formula* is a formula over unprimed and primed configuration variables without temporal operators, used for specifying constraints on two consecutive configurations. For an action formula φ_T, we can use $\Box\,\varphi_T$ to specify that a transition satisfying φ_T is taken at each timepoint. Conjoining this with a state formula φ_I specifying the set of initial configurations, we get the formula $\varphi_I \wedge \Box\,\varphi_T$ whose models correspond to executions of the transition system where φ_I specifies the set of initial configurations and φ_T specifies the transition relation.

Extended Syntax for Modeling Apart from the abbreviations already introduced, we will also use the following abbreviations to make our models more readable.

$\exists x' : \varphi$	$\triangleq$	$\exists K : \varphi'$ where φ' equals φ except that all occurrences of the form $x'[i]$ are replaced by $i \in K$, and K is a fresh second-order position variable
Enabled φ	$\triangleq$	$\exists\, x'_1, x'_2, \ldots, x'_n : \varphi$ where φ is an action formula, and $x_1, x_2, \ldots, x_n$ are all configuration variables occurring in φ
$x[i](v, v')$	$\triangleq$	$x[i] = v \wedge x'[i] = v'$

The formula **Enabled** φ is used to test if the transition represented by φ can be taken, and the formula $x[i](v, v')$ is used to say that the value of x changes from v to v' . Furthermore, we extend the range of configuration variables to any finite domain (rather than just boolean values) by using a standard encoding of a finite domain into a set of boolean variables. For example, when pc is a configuration variable representing a program counter, we can use $pc[i](5, 6)$ to express that the value of pc changes from 5 to 6.

To model the token passing protocol introduced in Section A.2, we use a configuration variable variable t where $t[i]$ is true iff process i has the token. The protocol is modeled by the formulas below. Note that if

$i = \$$, the token cannot be passed since there is no position $i + 1$.

$$
\begin{array}{lcl}
\textbf{initial} & = & \forall i : t[i] \longleftrightarrow i = 0 \\
\textbf{passtoken}(i) & = & \left[\begin{array}{l} \quad t[i] \ \wedge \ \neg t'[i] \ \wedge \ \neg t[i+1] \ \wedge \ t'[i+1] \\ \wedge \quad \forall j \notin \{i, i+1\} : t'[j] = t[j] \end{array}\right] \\
\textbf{transition} & = & \exists i : \textbf{passtoken}(i) \\
\textbf{idle} & = & \forall i : t'[i] = t[i] \\
\textbf{sys} & = & \textbf{initial} \ \wedge \ \Box \, (\textbf{transition} \ \vee \ \textbf{idle})
\end{array}
$$

The set of initial configurations, where the first process has the token, is specified by the state formula **initial**. The formula **passtoken**(i) specifies that the token is passed by process i to its neighbor, and the formula **idle** specifies that nothing happens. The formula **idle** is used to model that the system may do things between passing the token, and will be necessary for adequately modeling liveness properties. The transition relation is obtained by conjoining the action formulas **transition** and **idle**, which is combined with **initial** to form the system formula **sys**, representing executions of the system.

A model of the formula **sys** from the token passing example is given below:

$$
\begin{array}{ccccc}
\top & \bot & \bot & \bot & \bot \\
\bot & \top & \bot & \bot & \bot \\
\bot & \bot & \top & \bot & \bot \\
\bot & \bot & \top & \bot & \bot \\
\bot & \bot & \top & \bot & \bot \\
\bot & \bot & \bot & \top & \bot \\
\bot & \bot & \bot & \bot & \top \\
\bot & \bot & \bot & \bot & \top \\
\bot & \bot & \bot & \bot & \top \\
 & & \vdots & &
\end{array}
$$

A.4.2 Fairness

To verify liveness properties, we need to add *fairness assumptions*. In this paper, we use *weak fairness*, although the logic can be used to express other kinds of fairness assumptions as well, e.g., strong fairness. Weak fairness is specified on an action formula, and can be defined as

$$WF(\varphi_T) = \Box\Diamond \, (\varphi_T \ \vee \ \neg \textbf{Enabled} \ \varphi_T)$$

which states that the action specified by the formula φ_T is either taken infinitely often or disabled infinitely often. When specifying fairness for

concurrent systems, it is useful to specify weak fairness *for each process*, stating that each process that may execute will eventually do so. This is an assumtion on the scheduler of the system, assuring that the all processes in a system are scheduled infinitely often. We call this *process fairness*, and express it as:

$$\forall i : WF(\varphi_T(i))$$

where $\varphi_T(i)$ specifies all transitions in which process i is active. In the token passing example, we add process fairness to the transitions specified by **passtoken**(i) using the formula:

$$\varphi_{fair} = \forall i : WF(\textbf{passtoken}(i))$$

Let us expand the defintions to demonstrate the meaning of φ_{fair}. Substituting the definition of **Enabled** we obtain the formula

$$\forall i : \Box\Diamond \left[\begin{array}{lll} & \textbf{passtoken}(i) & \\ \vee & \neg\exists K : \exists j : \left[\begin{array}{ll} & j = i+1 \\ \wedge & t[i] \wedge i \notin K \wedge \neg t[j] \wedge j \in K \\ \wedge & \forall k \notin \{i,j\} : k \in K \longleftrightarrow t[k] \end{array} \right] \end{array} \right]$$

which after removal of the existential quantifier on K (an interpretation of K will always exist provided the other conditions hold) becomes:

$$\forall i : \Box\Diamond \left[\begin{array}{ll} & \textbf{passtoken}(i) \\ \vee & \neg\exists j : j = i+1 \wedge t[i] \wedge \neg t[j] \end{array} \right]$$

meaning that for all processes i, it is infinitely often the case that the token is passed *or* the token cannot be passed either because it is the rightmost process (no j exists such that $j = i+1$), the process does not have the token ($t[i]$ is false), or the neighboring process already has a token ($t[j]$ is true).

A.4.3 Specifying and Checking Properties

Let

$$\varphi_I \wedge \Box\varphi_T \wedge \varphi_{fair}$$

be a system specified with fairness assumptions. A *property* is given as a formula φ; for instance, an invariant property is of the form $\Box\varphi_{Inv}$ for a state formula φ_{Inv}. To check whether the system model satisfies the property φ, we check whether the formula

$$\varphi_I \wedge \Box\varphi_T \wedge \varphi_{fair} \wedge \neg\varphi$$

is satisfiable. If φ is a safety property, the fairness assumptions φ_{fair} are not necessary, and can be omitted.

Continuing the token passing example, we can check that there is never more than one token in the system by searching for models of the formula

$$\textbf{initial} \wedge \Box\,(\textbf{transition} \vee \textbf{idle}) \wedge \neg\Box\,\neg\exists i, j\ i \neq j \wedge t[i] \wedge t[j]$$

and whether the rightmost process must eventually get the token searching for models of the formula

$$\begin{array}{ll} & \textbf{initial} \\ \wedge & \Box\,(\textbf{transition} \vee \textbf{idle}) \\ \wedge & \forall i : \Box\Diamond\,(\textbf{transition}(i) \vee \neg\textbf{Enabled transition}(i)) \\ \wedge & \neg\Diamond\,\exists i : i = \$ \wedge t[i]\,. \end{array}$$

In the following sections, we discuss how to model parameterized algorithms and algorithms with different kinds of datatypes in our logic.

A.5 Parameterized Systems

Consider a system parameterized by the number of processes. Typical examples are algorithms designed to work for an arbitrary number of processes. In this case, we want to verify the system regardless of the number of processes.

We assume that the processes are homogeneous, i.e., that all processes have the same set of local states. We use a configuration variable x so that the value of $x[i]$ represents the local state of process i.

Local transitions, where a process can change local state from q to q' independently of other processes, can be expressed as

$$\exists i : x[i](q, q') \wedge \forall j \neq i : x[j] = x'[j]\ .$$

Other transitions need global conditions, for example that all processes at a position with a lower index should be in a particular state, say q_g. We can express this as

$$\exists i : x[i](q, q') \wedge (\forall j < i : x[j] = q_g) \wedge \forall j \neq i : x[j] = x'[j]\ .$$

We can also model transitions representing communication between two processes, e.g.,

$$\exists i : x[i](q, q') \wedge x[i+1](r, r') \wedge \forall j \notin \{i, i+1\} : x[j] = x'[j]\ .$$

We illustrate this type of representation using a number of examples.

Idle:	$ticket_i := 1 + \max_j ticket_j$
Waiting:	await $\forall j \neq i : (ticket_i < ticket_j \vee ticket_j = 0)$
Critical:	$ticket_i := 0$

Figure A.2: Bakery algorithm

A.5.1 The Bakery Algorithm

In the bakery algorithm for mutual exclusion due to Lamport [Lam74], there are an arbitrary number of processes waiting to get a "ticket" to get into the critical section. Each process that wants to get into the critical section receives a ticket which is the maximum of all the outstanding tickets plus one. When a process has the lowest outstanding ticket, it enters the critical section and drops the ticket when leaving. The algorithm is shown in Fig. A.2, where $ticket_i$ is used to denote the ticket value of process i or 0 if it does not have a ticket.

To model the bakery algorithm in $LTL(MSO)$, we change the perspective: rather than modeling the vector of process states, we let a configuration represent the states of the sequence of ticket numbers, using the configuration variable q. For each i, the value of $q[i]$ is

- $\perp$ if there is no process that has ticket $i + 1$,
- W if some process with ticket $i + 1$ is Waiting, and
- C if some process with ticket $i + 1$ is in Critical.

Note that we do not model tickets with number 0, since this is the ticket number of all "inactive" processes, and that ticket $i+1$ is modeled by $q[i]$. We implicitly use the invariant that each positive ticket number can be held by at most one process. This invariant can be verified separately, or not be assumed (for example by adding one more value of $q[i]$ representing that several processes have this ticket number).

The initial configuration and transition relation of the bakery algorithm can then be specified by the formulas shown in Fig. A.3.

We use the auxiliary formula **maxplusone**(i) to specify that i refers to the position representing next ticket, i.e., the maximum ticket number plus one, and the auxiliary formula **min**(i) to specify that i refers to the position representing the ticket that is next in line, i.e., the ticket with the minimum ticket number.

We use one action formula for the transition between states: **ticket**(i) specifies the transitions from $\perp$ to W, allowing ticket number $i+1$ to be taken, **enter**(i) specifies the transition from W to C, allowing a process with ticket number $i + 1$ to proceed to the critical section, and finally

$\mathbf{maxplusone}(i)$	$=$	$(i \neq 0 \longrightarrow q[i-1] \neq \bot) \wedge \forall j > i : q[j] = \bot$
$\mathbf{min}(i)$	$=$	$q[i] \neq \bot \wedge \forall j < i : q[j] = \bot$
$\mathbf{ticket}(i)$	$=$	$q[i](\bot, W) \wedge \mathbf{maxplusone}(i)$
$\mathbf{enter}(i)$	$=$	$q[i](W, C) \wedge \mathbf{min}(i)$
$\mathbf{exit}(i)$	$=$	$q[i](C, \bot)$
$\mathbf{copy}(i)$	$=$	$q[i] = q'[i]$
$\mathbf{idle}$	$=$	$\forall i : \mathbf{copy}(i)$
$\mathbf{a}(i)$	$=$	$(\mathbf{ticket}(\mathrm{i}) \vee \mathbf{enter}(i) \vee \mathbf{exit}(i)) \wedge$ $\forall j \neq i : \mathbf{copy}(j)$
$\mathbf{initial}$	$=$	$\forall i : q[i] = \bot$
$\mathbf{sys}$	$=$	$\mathbf{initial} \wedge \Box(\exists i : \mathbf{a}(i) \vee \mathbf{idle})$

Figure A.3: Bakery algorithm in *LTL*(*MSO*)

$\mathbf{exit}(i)$ specifies the transitions from C to $\bot$, allowing a process with ticket number $i+1$ to leave the critical section and return the ticket.

The system is specified by the formula **sys** which is the conjunction of the formula **initial** specifying the set of initial configurations and the formula $\Box(\exists i : \mathbf{a}(i) \vee \mathbf{idle})$ specifying that in each step either some action $\mathbf{a}(i)$ is taken by process i, or the system idles. The idle transitions are needed to verify liveness properties.

Mutual exclusion can be specified by the formula

$$\mathbf{mutex} \quad = \quad \Box \neg(\exists i : \exists j : i \neq j \wedge q[i] = C \wedge q[j] = C) \ .$$

In order to specify non-starvation, we add a fairness assumption for the actions $\mathbf{enter}(i)$ and $\mathbf{exit}(i)$. We add no fairness assumption for $\mathbf{ticket}(i)$, since the arrival of new processes should not be controlled by the algorithm itself.

$\mathbf{faira}(i)$	$=$	$(\mathbf{enter}(i) \vee \mathbf{exit}(i)) \wedge (\forall j \neq i : \mathbf{copy}(j))$
$\mathbf{fairness}$	$=$	$\forall i : \Box\Diamond(\mathbf{faira}(i) \vee \neg\mathbf{Enabled}(\mathbf{faira}(i)))$
$\mathbf{non\text{-}starvation}$	$=$	$\forall i : \Box(q[i] = W \longrightarrow \Diamond q[i] = C)$

To check that the algorithm satisfies mutual exclusion and non-starvation, we check whether the formulas

$$\mathbf{sys} \wedge \neg\mathbf{mutex}$$
$$\mathbf{sys} \wedge \mathbf{fairness} \wedge \neg\mathbf{non\text{-}starvation}$$

have any models.

The property that models are of arbitrary but fixed size implies that we actually verify the algorithm under the assumption that there is an

```
1:   await ∀j : j ≠ i : ¬s[j]
2:   w[i], s[i] := true, true
3:   if ∃j : j ≠ i : (pc[j] ≠ 1) ∧ (¬w[i])
         then s[i] := false ; goto 4
         else w[i] := false ; goto 5
4:   await ∃j : j ≠ i : s[j] ∧ ¬w[j] then w[i], s[i] := false, true
5:   await ∀j : j ≠ i : ¬w[j]
6:   await ∀j : j < i : ¬s[j]
7:   s[i] := false ; goto 1
```

Figure A.4: Szymanski's algorithm

arbitrarily chosen upper bound on the number of tickets in use at any time. For safety properties, this is not a limitation since violations will be finite sequences of execution steps, but for fairness assumptions it can play a role. For the bakery algorithm, it can be seen that an arbitrary upper limit on ticket numbers does not affect non-starvation for waiting processes, but in general one must be aware of this modeling constraint.

A.5.2 Szymanski's Algorithm

In the previous example there were an arbitrary number of processes, but there was a complete symmetry between the processes. In this example we will look at another algorithm that works for an arbitrary number of processes, but with the difference that they are organized in a linear array and thus will not be completely symmetric with respect to each other.

In Szymanski's algorithm for mutual exclusion [Szy90, GZ98], there are an arbitrary number of processes organized in a linear array, where the index of the array denotes the process ID. In the algorithm, the local state of each process i consists of a control state $pc[i]$, ranging over the integers from 1 to 7 and of two boolean flags, $w[i]$ and $s[i]$. A process i is in the critical section when its control state $pc[i]$ is equal to 7. We model this using three variables named pc, and w, and s, ranging over an array of the same length as the number of processes. The behavior for each process i is given in Fig. A.4, expressed in pseudo-code where the lines are numbered with the value of the control state pc. The version considered here is an idealized version. In most implementations a global guard (such as, e.g., $\forall j : j < i : s[j]$) is not atomic: in a more refined description of the algorithm this is a loop which checks the states of other processes.

For instance, according to the statement at line 6, if the control state of a process i is 6, and the value of s is *false* in all processes with a

lower index (i.e., for all processes j with $j < i$), then the control state of process i may be changed to 7. In a similar manner, according to the statement at line 4, if the control state of a process i is 4, and if there is at least another process j (either with a lower index or a higher index than i) where the value of $s[j]$ is *true* and the value of $w[j]$ is *false*, then the control state, $w[i]$, and $s[i]$, in i may be changed to 5, *false*, and *true*, respectively.

The full model in $LTL(MSO)$ is given in Fig. A.5. Auxiliary predicates **copy**, **copy-w**, **copy-s** and **copy-other** have been added to denote that some variables are not affected by the transition. The action formulas $\mathbf{a1}(i)$ through $\mathbf{a7}(i)$ are used to specify the transitions in the algorithm. To see how the above statements are modeled, line 1 can for example be modeled by the following formula:

$$\exists i : pc[i](1,2) \ \wedge \ (\forall j : j \neq i : \neg s[j]) \ \wedge \ w'[i] = w[i] \ \wedge \ s'[i] = s[i]$$

where the difference to line 1 is mainly that the program counter pc is made explicit.

Like in the Bakery algorithm in Section A.5.1, we add a system formula **sys** by conjoining the formula **initial** specifying the set of initial configurations and the formulas for the transitions of the algorithm. The formulas **safety** for verifying mutual exclution and **liveness** for verifying non-starvation are also written in a similar way.

A.5.3 Dijkstra's Algorithm

In Fig. A.6, we show an idealized version of Dijkstra's protocol [LPS93] for ensuring mutual exclusion among an arbitrary number of processes. Each process i has a control state ranging over the integers from 1 to 7 and a variable $flag[i]$ ranging over $\{0, 1, 2\}$. Furthermore, a global variable p ranging over process indices is used. In the algorithm, line 6 represents the critical section.

We model the global variable with a configuration variable p such that $p[i]$ is true iff the global variable p points to process i. The resulting $LTL(MSO)$ model is given in Fig. A.7.

A.5.4 Burns's Algorithm

Burns's mutual exclusion algorithm [LPS93] is given in Fig. A.8. Each process i has a control state ranging over the integers from 1 to 7 and a variable $flag[i]$ ranging over $\{0, 1\}$. The critical section is represented by line 6.

We model the values 0 and 1 with the booleans such that 0 is false and 1 is true. The $LTL(MSO)$ model for the algorithm is given in Fig. A.9.

$\mathbf{copy}(i)$	$=$	$pc[i] = pc'[i] \wedge w[i] = w'[i] \wedge s[i] = s'[i]$
idle	$=$	$\forall i : \mathbf{copy}(i)$
$\mathbf{copy\text{-}w}(i)$	$=$	$w[i] = w'[i]$
$\mathbf{copy\text{-}s}(i)$	$=$	$s[i] = s'[i]$
$\mathbf{copy\text{-}other}(i)$	$=$	$(\forall j \neq i : \mathbf{copy}(j))$
$\mathbf{a1}(i)$	$=$	$pc[i](1,2) \wedge (\forall j \neq i : \neg s[j]) \wedge$ $\mathbf{copy\text{-}w}(i) \wedge \mathbf{copy\text{-}s}(i)$
$\mathbf{a2}(i)$	$=$	$pc[i](2,3) \wedge w'[i] \wedge s'[i]$
$\mathbf{a3a}(i)$	$=$	$pc[i](3,4) \wedge \neg s'[i] \wedge \mathbf{copy\text{-}w}(i) \wedge$ $\exists j \neq i : \neg(pc[j] = 1) \wedge \neg w[j]$
$\mathbf{a3b}(i)$	$=$	$pc[i](3,5) \wedge \neg w'[i] \wedge \mathbf{copy\text{-}s}(i) \wedge$ $\neg(\exists j \neq i : \neg(pc[j] = 1) \wedge \neg w[j])$
$\mathbf{a3}(i)$	$=$	$\mathbf{a3a}(i) \vee \mathbf{a3b}(i)$
$\mathbf{a4}(i)$	$=$	$pc[i](4,5) \wedge \neg w'[i] \wedge s'[i] \wedge$ $(\exists j \neq i : s[j] \wedge \neg w[j])$
$\mathbf{a5}(i)$	$=$	$pc[i](5,6) \wedge (\forall j \neq i : \neg w[j]) \wedge$ $\mathbf{copy\text{-}w}(i) \wedge \mathbf{copy\text{-}s}(i)$
$\mathbf{a6}(i)$	$=$	$pc[i](6,7) \wedge (\forall j < i : \neg s[j]) \wedge$ $\mathbf{copy\text{-}w}(i) \wedge \mathbf{copy\text{-}s}(i)$
$\mathbf{a7}(i)$	$=$	$pc[i](7,1) \wedge \neg s'[i] \wedge \mathbf{copy\text{-}w}(i)$
$\mathbf{a}(i)$	$=$	$\mathbf{a1}(i) \vee \mathbf{a2}(i) \vee \mathbf{a3}(i) \vee \mathbf{a4}(i) \vee$ $\mathbf{a5}(i) \vee \mathbf{a6}(i) \vee \mathbf{a7}(i)$
initial	$=$	$\forall i : pc[i] = 1$
sys	$=$	$\mathbf{initial} \wedge$ $\Box(\exists i : (\mathbf{a}(i) \wedge \mathbf{copy\text{-}other}(i)) \vee \mathbf{idle})$
fairness	$=$	$\forall i : \Box\Diamond(\mathbf{a}(i) \vee \neg\mathbf{Enabled}(\mathbf{a}(i)))$
mutex	$=$	$\Box\neg\exists i : \exists j : i \neq j \wedge pc[i] = 7 \wedge pc[j] = 7$
non-starvation	$=$	$\forall i : \Box\,(pc[i] = 2 \longrightarrow \Diamond pc[i] = 7)$
safety	$=$	$\mathbf{sys} \wedge \neg\mathbf{mutex}$
liveness	$=$	$\mathbf{sys} \wedge \mathbf{fairness} \wedge \neg\mathbf{non\text{-}starvation}$

Figure A.5: Szymanski's algorithm in $LTL(MSO)$

```
1:    flag[i] := 1
2:    if p ≠ i then
          await flag[p] = 0 then
3:            p := i
4:    flag[i] := 2
5:    if ∃j ≠ i : flag[j] = 2 then goto 1
6:    flag[i] := 0 ; goto 1
```

Figure A.6: Dijkstra's algorithm

A.5.5 A Termination Detection Algorithm

We can also models ring shaped parameterized systems in our framework, which we illustrate with an algorithm for termination detection among an arbitrary number of processes organized in a ring shaped network, due to Dijkstra et al. [DFvG83]. The algorithm uses a colored token which is passed around the ring to check that all processes in the ring have terminated.

A process can either be *non-idle* or *idle*. When all processes are idle, we say that the system has terminated. A process can spontaneously change its state from non-idle to idle, i.e., it terminates. To detect that all processes are idle, a designated processes sends out a token which it colors *white*. When the token is passed to the next processes, the process passing the token paints it black if it is non-idle. When the token comes back to the process which sent out the token, it is white if the system has terminated, and black otherwise.

The system can be modeled by numbering the processes from 0 to $n-1$ and using three arrays holding three local variables the processes. Only process 0 may initiate the algorithm by sending out a new token. The variables are $q[i]$ which is true iff process i is idle, and $t[i]$ ranging over $\{\mathbf{black}, \mathbf{white}, \mathbf{none}\}$, which has the value **none** when process i does *not* have the token, and otherwise denotes the color of the token. In addition, process 0 has a boolean variable w, which is true if it has stayed idle during the current round. The value of w is only relevant for process 0.

Initially, we have $q[i] = \mathit{false}$ for all i, and $t[0] = \mathbf{black}$, and $t[i] = \mathbf{none}$ for all $0 < i < n$, and $w = \mathit{false}$. The algorithm can be described by the statements in Fig. A.10, for each process i.

The three first types of statements describe the underlying computation: a process can become idle autonomously (first statement), and it can become non-idle if its predecessor is non-idle (second statement). In addition (third statement), process 0 must set w to *false* if it becomes non-idle. The fourth statement starts a round of the detection algorithm.

copy(i)	=	$pc[i] = pc'[i] \wedge flag[i] = flag'[i] \wedge p[i] = p'[i]$
copy-flag(i)	=	$flag[i] = flag'[i]$
copy-p	=	$\forall k : p[k] = p'[k]$
copy-other(i)	=	$\forall j \neq i :$ **copy**(j)
idle	=	$\forall i :$ **copy**(i)
set-p(i)	=	$\forall j : p'[j] \longleftrightarrow j = i$
zeropflag	=	$\forall k : (p[k] \dashrightarrow flag[k] = 0)$
a1(i)	=	$pc[i](1,2) \wedge flag'[i] = 1 \wedge$ **copy-p**
a2a(i)	=	$pc[i](2,3) \wedge \neg p[i] \wedge$ **zeropflag** $\wedge$ **copy-p**
a2b(i)	=	$pc[i](2,4) \wedge p[i] \wedge$ **copy-flag**$(i) \wedge$ **copy-p**
a2(i)	=	**a2a**$(i) \vee$ **a2b**(i)
a3(i)	=	$pc[i](3,4) \wedge$ **set-p**$(i) \wedge$ **copy-flag**(i)
a4(i)	=	$pc[i](4,5) \wedge flag'[i] = 2 \wedge$ **copy-p**
a5a(i)	=	$pc[i](5,1) \wedge$ **copy-flag**$(i) \wedge$ **copy-p** $\wedge$ $\exists j \neq i : flag[j] = 2$
a5b(i)	=	$pc[i](5,6) \wedge$ **copy-flag**$(i) \wedge$ **copy-p** $\wedge$ $\neg\exists j \neq i : flag[j] = 2$
a5(i)	=	**a5a**$(i) \vee$ **a5b**(i)
a6(i)	=	$pc[i](6,1) \wedge flag'[i] = 0 \wedge$ **copy-p**
a(i)	=	**a1**$(i) \vee$ **a2**$(i) \vee$ **a3**$(i) \vee$ **a4**$(i) \vee$ **a5**$(i) \vee$ **a6**(i)
initial	=	$\forall i : pc[i] = 1 \wedge flag[i] = 0 \wedge \neg p[i]$
sys	=	**initial** $\wedge$ $\Box(\exists i :$ (**a**$(i) \wedge$ **copy-other**$(i)) \vee$ **idle**$)$
fairness	=	$\forall i : \Box\Diamond($**a**$(i) \vee \neg$**Enabled**(**a**$(i)))$
mutex	=	$\Box \neg\exists i : \exists j : i \neq j \wedge pc[i] = 6 \wedge pc[j] = 6$
non-starvation	=	$\forall i : \Box\,(pc[i] = 1 \dashrightarrow \Diamond pc[i] = 6)$
safety	=	**sys** $\wedge \neg$**mutex**
liveness	=	**sys** $\wedge$ **fairness** $\wedge \neg$**non-starvation**

Figure A.7: Dijkstra's algorithm in *LTL*(*MSO*)

```
1:  flag[i] := 0
2:  if ∃j < i : flag[j] = 1 then goto 1
3:  flag[i] := 1
4:  if ∃j < i : flag[j] = 1 then goto 1
5:  await ∀j > i : flag[j] ≠ 1
6:  flag[i] := 0 ; goto 1
```

Figure A.8: Burns's algorithm

$\mathbf{copy}(i)$	$=$	$pc[i] = pc'[i] \wedge flag[i] = flag'[i]$
$\mathbf{copy\text{-}flag}(i)$	$=$	$flag[i] = flag'[i]$
$\mathbf{copy\text{-}other}(i)$	$=$	$\forall j \neq i : \mathbf{copy}(j)$
$\mathbf{idle}$	$=$	$\forall i : \mathbf{copy}(i)$
$\mathbf{a1}(i)$	$=$	$pc[i](1,2) \wedge \neg flag'[i]$
$\mathbf{a2a}(i)$	$=$	$pc[i](2,1) \wedge (\exists j < i : flag[j]) \wedge \mathbf{copy\text{-}flag}(i)$
$\mathbf{a2b}(i)$	$=$	$pc[i](2,3) \wedge (\neg\exists j < i : flag[j]) \wedge \mathbf{copy\text{-}flag}(i)$
$\mathbf{a2}(i)$	$=$	$\mathbf{a2a}(i) \vee \mathbf{a2b}(i)$
$\mathbf{a3}(i)$	$=$	$pc[i](3,4) \wedge flag'[i]$
$\mathbf{a4a}(i)$	$=$	$pc[i](4,1) \wedge (\exists j < i : flag[j]) \wedge \mathbf{copy\text{-}flag}(i)$
$\mathbf{a4b}(i)$	$=$	$pc[i](4,5) \wedge (\neg\exists j < i : flag[j]) \wedge \mathbf{copy\text{-}flag}(i)$
$\mathbf{a4}(i)$	$=$	$\mathbf{a4a}(i) \vee \mathbf{a4b}(i)$
$\mathbf{a5}(i)$	$=$	$pc[i](5,6) \wedge (\forall j > i : \neg flag[j]) \wedge \mathbf{copy\text{-}flag}(i)$
$\mathbf{a6}(i)$	$=$	$pc[i](6,1) \wedge \neg flag'[i]$
$\mathbf{a}(i)$	$=$	$\mathbf{a1}(i) \vee \mathbf{a2}(i) \vee \mathbf{a3}(i) \vee$
		$\mathbf{a4}(i) \vee \mathbf{a5}(i) \vee \mathbf{a6}(i)$
$\mathbf{initial}$	$=$	$\forall i : pc[i] = 1 \wedge flag[i] = 0$
$\mathbf{sys}$	$=$	$\mathbf{initial} \wedge$
		$\Box(\exists i : (\mathbf{a}(i) \wedge \mathbf{copy\text{-}other}(i)) \vee \mathbf{idle})$
$\mathbf{fairness}$	$=$	$\forall i : \Box\Diamond(\mathbf{a}(i) \vee \neg\mathbf{Enabled}(\mathbf{a}(i)))$
$\mathbf{mutex}$	$=$	$\Box\neg\exists i : \exists j : i \neq j \wedge pc[i] = 6 \wedge pc[j] = 6$
$\mathbf{non\text{-}starvation}$	$=$	$\forall i : \Box\,(pc[i] = 1 \longrightarrow \Diamond pc[i] = 6)$
$\mathbf{safety}$	$=$	$\mathbf{sys} \wedge \neg\mathbf{mutex}$
$\mathbf{liveness}$	$=$	$\mathbf{sys} \wedge \mathbf{fairness} \wedge \neg\mathbf{non\text{-}starvation}$

Figure A.9: Burns's algorithm in $LTL(MSO)$

- $q[i] := \mathit{true}$
- **if** $i > 0 \wedge \neg q[i-1]$ **then** $q[i] := \mathit{false}$
- **if** $\neg q[n-1]$ **then** $q[0], w := \mathit{false}, \mathit{false}$
- **if** $i = 0 \wedge q[0] \wedge (t[0] = \textbf{black} \vee \neg w)$ **then** $t[0], t[1], w := \textbf{none}, \textbf{white}, \mathit{true}$
- **if** $i < n-1 \wedge t[i] \neq \textbf{none} \wedge q[i]$ **then** $t[i], t[i+1] := \textbf{none}, t[i]$
- **if** $i = n-1 \wedge t[n-1] \neq \textbf{none} \wedge \neg q[n-1]$ **then** $t[n-1], t[0] := \textbf{none}, t[i]$
- **if** $i < n-1 \wedge t[i] \neq \textbf{none} \wedge \neg q[i]$ **then** $t[i], t[i+1] := \textbf{none}, \textbf{black}$
- **if** $i = n-1 \wedge t[n-1] \neq \textbf{none} \wedge \neg q[n-1]$ **then** $t[n-1], t[0] := \textbf{none}, \textbf{black}$

Figure A.10: A Termination Detection Algorithm

In the next two statements, a process just forwards the token if it is idle. Finally, in the last two statements, if a process is non-idle, the token is painted black and then forwarded. Note how the ring is modeled by allowing process $n-1$ to communicate with process 0.

The model is given in Fig. A.11. The formula **safety** is used to verify that if process 0 signals termination, then all processes are idle.

A.6 Communication Protocols

Our framework can be used to model queues and stacks by letting each position in the word represent a position in the queue or the stack. Integer variables can also be modeled, using the word to represent the digits of the word in some base. These data types are common in communication protocols, where processes communicate through a queue and integer variables can be used to model sequence numbers of the messages that are passed. We will use communication protocols to illustrate how we can represent these data types and operations on them.

Queues and Stacks Let us describe how to represent queues and stacks in our framework. We use a configuration variable q where $q[i]$ is the queue or stack content at position i. Since our transitions preserve the length of the words, we cannot dynamically create new positions. Therefore, to allow for a dynamic data structure, we add a *padding symbol* $\perp$ to represent empty slots. Recall that configurations are of arbitrary length, so even though we can not model unbounded queues, we can model arbitrary-length queues. The difference between unbounded

$\mathbf{copy}(i)$	$=$	$t[i] = t'[i] \wedge w[i] = w'[i] \wedge q[i] = q'[i]$
$\mathbf{copy\text{-}other}(i)$	$=$	$\forall j \neq i : \mathbf{copy}(j)$
$\mathbf{copy\text{-}other2}(i,j)$	$=$	$\forall k : \neg(k = i \vee k = j) \longrightarrow \mathbf{copy}(k)$
$\mathbf{copy\text{-}q}(i)$	$=$	$q[i] = q'[i]$
$\mathbf{copy\text{-}t}(i)$	$=$	$t[i] = t'[i]$
$\mathbf{idle}$	$=$	$\forall i : \mathbf{copy}(i)$
$\mathbf{move\text{-}token}(i,j)$	$=$	$t[i] = t'[j]$
$\mathbf{adjacent}(i,j)$	$=$	$j = i + 1 \vee (j = 0 \wedge i = \$)$
$\mathbf{pass}(i,j)$	$=$	$i \neq 0 \wedge t[i] \neq \mathbf{none} \wedge$
		$(\neg q[i] \longrightarrow t'[j] = \mathbf{black}) \wedge$
		$(q[i] \longrightarrow \mathbf{move\text{-}token}(i,j)) \wedge t'[i] = \mathbf{none} \wedge$
		$\mathbf{copy\text{-}q}(i) \wedge \mathbf{copy\text{-}q}(j) \wedge w[0] = w'[0]$
$\mathbf{start}(i,j)$	$=$	$i = 0 \wedge q[i] \wedge (t[i] = \mathbf{black} \vee \neg w[0]) \wedge$
		$t'[i] = \mathbf{none} \wedge t'[j] = \mathbf{white} \wedge w'[0] \wedge$
		$\mathbf{copy\text{-}q}(i) \wedge \mathbf{copy\text{-}q}(j)$
$\mathbf{comp1}(i)$	$=$	$q'[i] \wedge \mathbf{copy\text{-}t}(i) \wedge w[0] = w'[0]$
$\mathbf{comp2}(i,j)$	$=$	$\neg q[i] \wedge \mathbf{copy}(i) \wedge \mathbf{copy\text{-}t}(j) \wedge q'[j] \wedge$
		$(j = 0 \longrightarrow \neg w'[0]) \wedge (j \neq 0 \longrightarrow w[0] = w'[0])$
$\mathbf{a1}(i)$	$=$	$\mathbf{copy\text{-}other}(i) \wedge \mathbf{comp1}(i)$
$\mathbf{a2}(i)$	$=$	$\exists j : \mathbf{adjacent}(i,j) \wedge \mathbf{copy\text{-}other2}(i,j) \wedge$
		$(\mathbf{start}(i,j) \vee \mathbf{pass}(i,j) \vee \mathbf{comp2}(i,j))$
$\mathbf{a}(i)$	$=$	$\mathbf{a1}(i) \vee \mathbf{a2}(i)$
$\mathbf{initial}$	$=$	$\forall i : (i = 0 \longrightarrow t[i] = \mathbf{black} \wedge \neg w[i]) \wedge$
		$(i \neq 0 \longrightarrow t[i] = \mathbf{none}) \wedge \neg q[i]$
$\mathbf{sys}$	$=$	$\mathbf{initial} \wedge \Box(\exists i : \mathbf{a}(i) \vee \mathbf{idle})$
$\mathbf{termination}$	$=$	$\Box\,(\exists i = 0 : t[i] = \mathbf{white} \wedge w[0]) \longrightarrow \forall i : q[i]$
$\mathbf{safety}$	$=$	$\mathbf{sys} \wedge \neg\mathbf{termination}$

Figure A.11: A Termination Detection Algorithm in *LTL*(*MSO*)

and arbitrary length is important for liveness properties, but not for safety properties.

Below, we model sending and receiving a message denoted by the parameter m to and from a queue represented using a configuration variable denoted by the parameter q. Messages are sent by replacing the $\perp$ to the right of the rightmost message, and received by replacing the leftmost message by a $\perp$. The empty queue is described by $\mathbf{empty}(q)$.

$$\mathbf{send}(q, m) \quad = \quad \exists i : \left[\begin{array}{ll} & q'[i] = m \ \wedge \ q[i] = \perp \\ \wedge & \forall j \neq i : q[i] = q'[i] \\ \wedge & \forall j : i = j + 1 \longrightarrow q[j] \neq \perp \\ \wedge & \forall j > i : q[j] = \perp \end{array}\right]$$

$$\mathbf{receive}(q, m) \quad = \quad \exists i : \left[\begin{array}{ll} & q'[i] = \perp \ \wedge \ q[i] = m \\ \wedge & \forall j \neq i : q[i] = q'[i] \\ \wedge & \forall j < i : q[j] = \perp \end{array}\right]$$

$$\mathbf{empty}(q) \quad = \quad \forall i : q[i] = \perp$$

For stacks, we model the push and pop operations below. The stack grows from left to right. The empty stack is described by $\mathbf{empty}(q)$.

$$\mathbf{push}(q, m) \quad = \quad \left[\begin{array}{ll} & q'[i] = m \ \wedge \ q[i] = \perp \\ \wedge & \forall j \neq i : q[i] = q'[i] \\ \wedge & \forall j : i = j + 1 \longrightarrow q[j] \neq \perp \\ \wedge & \forall j > i : q[j] = \perp \end{array}\right]$$

$$\mathbf{pop}(q, m) \quad = \quad \left[\begin{array}{ll} & q'[i] = \perp \ \wedge \ q[i] = m \\ \wedge & \forall j \neq i : q[i] = q'[i] \\ \wedge & \forall j : i = j + 1 \longrightarrow q[j] \neq \perp \\ \wedge & \forall j > i : q[j] = \perp \end{array}\right]$$

$$\mathbf{empty}(q) \quad = \quad \forall i : q[i] = \perp$$

We model sends to *lossy channels*, where messages may be lost, with the formula $\mathbf{lossend}(q, m)$ defined as

$$\mathbf{send}(q, m) \ \vee \ \forall i : q[i] = q'[i]$$

i.e., the message can be lost immediately when sending.

Integers Integer variables can be represented in many ways using a word. One alternative is to use a binary encoding of the integer value,

such that the word represents the value of the integer variable in binary with the most significant bit to the left. This has the advantage that addition and multiplication can be performed using a regular transition relation. For example, if we use the configuration variable x and y to represent two numbers, the operation $x := x + y$ can be modeled by the formula

$$\exists C : \left[\begin{array}{lll} & \$ \notin C & \\ \wedge & \forall i : (x'[i] \longleftrightarrow x[i]) \longleftrightarrow (y[i] \longleftrightarrow i \in C) & \\ \wedge & \forall i : i - 1 \in C \longleftrightarrow \left[\begin{array}{ll} & (x[i] \wedge y[i]) \\ \vee & (x[i] \wedge i \in C) \\ \vee & (y[i] \wedge i \in C) \end{array}\right] & \end{array}\right]$$

The second-order variable C is used to implement a carry-bit in the addition. The formula consists of three conjuncts. The first sets the carry-bit to false in the last position, corresponding to the least significant bit. The second part adds $x[i]$ and $y[i]$ and the carrybit $i \in C$, putting the result in $x'[i]$ (to see this, note that $(\varphi_1 \longleftrightarrow \varphi_2) \longleftrightarrow (\varphi_3 \longleftrightarrow \varphi_4)$ is true iff an even number of the formulas $\varphi_1, \varphi_2, \varphi_3, \varphi_4$ are true). The last part updates the carry-bit for $i - 1$ in case there was an overflow.

The binary encoding works well when the system consists only of integer variables and has been used for the verification of numerous examples, for example in the tool LASH [BFL]. When integer variables are used in combination with other datatypes, for example as a process index or a sequence number in a communication protocol, it can be more natural to use a *unary encoding*. With this encoding, addition and multiplication can not be expressed as a regular transition relation, but operations relating the variable with the other datatypes, for example changing the state of a process pointed to by a process index variable, can be performed.

In the following subsections, we model two communication protocols using the encodings of data types described above.

A.6.1 The Alternating Bit Protocol

We illustrate encoding of queues in our framework with the well-known Alternating Bit Protocol [BSW69], a protocol used for delivering messages over unbounded channels which are faulty in the sense that they may lose messages but not reorder them.

There are two channels, one for sending messages from the sender to the receiver, and one for sending acknowledgments from the receiver to the sender. Each message is given a sequence number and the sender waits for an acknowledgment from the receiver before sending a new message. Until this acknowledgment is received, the sender may resend

Sender	
1:	**protsend**
2:	(**lossend**$(msg, 0)$ **OR** **receive**$(ack, 1)$) ; **goto** 2 **OR** **receive**$(ack, 0)$
3:	**protsend**
4:	(**lossend**$(msg, 1)$ **OR** **receive**$(ack, 0)$) ; **goto** 4 **OR** **receive**$(ack, 1)$; **goto** 1

Receiver	
1:	(**lossend**$(ack, 1)$ **OR** **receive**$(msg, 1)$) ; **goto** 1 **OR** **receive**$(msg, 0)$
2:	**protreceive**
3:	(**lossend**$(ack, 0)$ **OR** **receive**$(msg, 0)$) ; **goto** 3 **OR** **receive**$(msg, 1)$
4:	**protreceive** ; **goto** 1

Figure A.12: The Alternating Bit Protocol

the message. When the receiver has acknowledged the message, the procedure is repeated but with the sequence number inverted. Both the sender and the receiver ignore messages with unexpected sequence numbers.

To model the service provided by the protocol, we consider two operations **protsend** and **protreceive**, modeling calls from the upper layers of the protocols. Thus, **protsend** denotes that there is a new message from the sender side, and **protreceive** denotes that the receiver side signals that a message has been received. We denote the two channels *msg* and *ack*, where *msg* is the channel used for messages and *ack* is the channel used for acknowledgments.

A high level description for the sender and the receiver is given in Fig. A.12. The notation S **OR** S' means that either S or S' is executed, but not both of them.

One property of the algorithm specifies that the operations **protsend** and **protreceive** alternate after each other such that the two operations never occur consecutively. We model this by adding an *observer* that records the last operation (**protsend** or **protreceive**) initialized to **protreceive** and checks that a **protsend** operation can not occur

when the observer is in state **protsend** and similarly that a **protreceive** operation can not occur when the observer is in state **protreceive**.

An $LTL(MSO)$ model of the Alternating Bit Protocol is given in Fig. A.13.

A.6.2 A Sliding Window Protocol

We illustrate the use of integers with a sliding window protocol (for a general description on sliding window protocols, see, e.g., Tannenbaum [Tan96] Ch. 3). Like the Alternating Bit Protocol, the protocol is intended to provide reliable transmission of messages across an unreliable channel.

The sender and receiver employ a so-called sliding window protocol, in which messages sent over the channel are provided with a sequence number, assigned in a cyclic fashion from 0 to $max - 1$ and then starting at 0 again. The receiver acknowledges messages using a separate channel, which we model with a direct communication between the receiver and the sender.

Initially, the sender transmits messages with consecutive sequence numbers $0, 1, 2$, etc. Since the channel may lose messages, the sender cannot know whether the messages will reach the receiver. Therefore, the sender also waits for acknowledgments from the receiver. An acknowledgment with sequence number n signals that the receiver has correctly received messages up to sequence number $n - 1$. There must never be more than $max - 1$ outstanding messages. Therefore, after sending messages 0 through $max - 2$, the sender must wait for an acknowledgment. After receiving an acknowledgment for a message, say 3, the sender may continue to send messages $max - 1$, 0, and 1. If no acknowledgment arrives for any outstanding messages, it is assumed to be lost and the sender should resend outstanding messages after some period of time.

The range of sequence number representing the outstanding messages is called the *sender window* and is modeled by two variables *low* and *high*, where the outstanding messages have sequence numbers n with $low \leq n < high$, if $low \leq high$, and with $low \leq n$ or $n < high$, if $high < low$. The integer variable *next* denotes the sequence number of the next message the receiver expects to receive. A high level version of the protocol is given in Fig. A.14, where addition is performed modulo *max*.

We model this protocol in $LTL(MSO)$ with a configuration variable for each of the integer variables with the same name. The formula $low[i]$ will be true if and only if the integer variable *low* is equal to i. The channel will be limited to a fixed capacity (say 3). Since the messages contains arbitrary sequence numbers and we have a finite alphabet, we can not model a channel of arbitrary size. Instead, we use three configuration

copy(x)	$=$	$\forall i : x[i] = x'[i]$
copy-other(x, i)	$=$	$\forall j \neq i : x[j] = x'[j]$
copy-channels	$=$	**copy**$(msg) \wedge$ **copy**(ack)
idle	$=$	**copy-channels** $\wedge$ **copy**$(pc) \wedge$ **copy**(obs)
observe(v)	$=$	$obs'[0] = v \wedge$ **copy-other**(obs, i)
sa1	$=$	$pc[0](1, 2) \wedge$ **copy-other**$(pc, 0) \wedge$ **copy-channels** $\wedge$ **observe**(**protsend**)
sa2a	$=$	$pc[0](2, 2) \wedge$ **copy-other**$(pc, 0) \wedge$ **lossend**$(msg, 0) \wedge$ **copy**$(ack) \wedge$ **copy**(obs)
sa2b	$=$	$pc[0](2, 2) \wedge$ **copy-other**$(pc, 0) \wedge$ **receive**$(ack, 1) \wedge$ **copy**$(msg) \wedge$ **copy**(obs)
sa2c	$=$	$pc[0](2, 3) \wedge$ **copy-other**$(pc, 0) \wedge$ **receive**$(ack, 0) \wedge$ **copy**$(msg) \wedge$ **copy**(obs)
sa3	$=$	$pc[0](3, 4) \wedge$ **copy-other**$(pc, 0) \wedge$ **copy-channels** $\wedge$ **observe**(**protsend**)
sa4a	$=$	$pc[0](4, 4) \wedge$ **copy-other**$(pc, 0) \wedge$ **lossend**$(msg, 1) \wedge$ **copy**$(ack) \wedge$ **copy**(obs)
sa4b	$=$	$pc[0](4, 4) \wedge$ **copy-other**$(pc, 0) \wedge$ **receive**$(ack, 0) \wedge$ **copy**$(msg) \wedge$ **copy**(obs)
sa4c	$=$	$pc[0](4, 1) \wedge$ **copy-other**$(pc, 0) \wedge$ **receive**$(ack, 1) \wedge$ **copy**$(msg) \wedge$ **copy**(obs)
sender	$=$	**sa1** $\vee$ **sa2a** $\vee$ **sa2b** $\vee$ **sa2c** $\vee$ **sa3** $\vee$ **sa4a** $\vee$ **sa4b** $\vee$ **sa4c**
receiver	$=$	*Defined similarly as sender with* $pc[1]$ *instead of* $pc[0]$ *and observing* **protreceive**
a	$=$	**sender** $\vee$ **receiver**
initial	$=$	$pc[0] = 1 \wedge pc[1] = 1 \wedge$ **empty**$(msg) \wedge$ **empty**$(ack) \wedge$ $obs[0] =$ **protreceive**
sys	$=$	**initial** $\wedge \Box($**a** $\vee$ **idle**$)$
receivealt	$=$	$\Box$ ($obs[0] =$ **protreceive** $\longrightarrow \neg($**ra2** $\vee$ **ra4**$)$)
sendalt	$=$	$\Box$ ($obs[0] =$ **protsend** $\longrightarrow \neg($**sa1** $\vee$ **sa3**$)$)
safety	$=$	**sys** $\wedge \neg$**sendalt** $\wedge \neg$**receivealt**

Figure A.13: The Alternating Bit Protocol in $LTL(MSO)$

Initially, $low = 0$, $next = 0$, and $high = 0$.

- (enlarge window) **if** $low \neq high + 1$ **then** $high := high + 1$
- (send) **for any** n **if** $\left[\begin{array}{ll} & (low \leq high \longrightarrow low \leq n \wedge n < high) \\ \wedge & (high < low \longrightarrow low \leq n \vee n < high) \end{array}\right]$ **then send**(c, n)
- (receive) **receive**$(c, next); next := next + 1$
- (synchronous ack) $low := next$

Figure A.14: A Sliding Window Protocol

variables c_1,c_2 and c_3 where $c_k[i]$ is true if and only if position k in the channel contains a message with sequence number i.

The full $LTL(MSO)$ model is given in Fig. A.15. The formula **a1** corresponds to enlarging the window, the formula **a2** to sending a message, the formula **a3** to receiving a message, the formulas **a4** and **a5** to movement within the channel and the formula **a6** to a synchronous ack.

The safety property **inside-window** specifies that the receiver is never outside the sending window, which can be seen as a check that the protocol synchronizes correctly.

A.7 Büchi Normal Form

In this section, we describe how to transform a *restricted* formula in $LTL(MSO)$ into an equivalent formula in *Büchi Normal Form*, defined as follows.

Definition A.1 ***(Büchi Normal Form)*** *A formula is in* Büchi Normal Form *if it is of the form*

$$\phi_I \wedge \Box\, \phi_T \wedge \Box\Diamond\, \phi_F$$

where the formulas ϕ_I, ϕ_T, ϕ_F are MSO formulas without temporal operators. □

Formulas in Büchi Normal Form correspond to *Büchi regular transition systems (BRTS)*, defined in Section A.8, which accept models of a formula. In this section, we show how to transform a formula in $LTL(MSO)$ into an equivalent formula in Büchi Normal Form.

The idea of the construction is to generalize the standard translation of propositional temporal logic to Büchi Automata [VW86, Var91] — the semantics of temporal operators is translated to additional state and transition information in the BRTS. In our case, temporal operators are translated to new configuration variables which represent the

$$
\begin{array}{lcl}
\textbf{copy}(x) & = & \forall i : x[i] = x'[i] \\
\textbf{copy-other}(x,i) & = & \forall j \neq i : x[j] = x'[j] \\
\textbf{copy-channel} & = & \textbf{copy}(c_1) \wedge \textbf{copy}(c_2) \wedge \textbf{copy}(c_3) \\
\textbf{copy-proc} & = & \textbf{copy}(low) \wedge \textbf{copy}(high) \wedge \textbf{copy}(next) \\
\textbf{idle} & = & \textbf{copy-channel} \wedge \textbf{copy-proc} \\
\textbf{adjacent}(i,j) & = & j = i+1 \vee (j = 0 \wedge i = \$) \\
\textbf{between}(i,j,k) & = & (i \leq k \longrightarrow i \leq j \wedge j < k) \wedge \\
 & & (k < i \longrightarrow i \leq j \vee j < k) \\
\textbf{addone}(x) & = & \exists p,q : \textbf{adjacent}(p,q) \wedge \\
 & & x[p] \wedge \neg x'[p] \wedge x'[q] \wedge \\
 & & \forall r : r = p \vee r = q \vee (\neg x[r] \wedge \neg x'[r]) \\
\textbf{allfalse}(x) & = & \forall i : \neg x[i] \\
\textbf{a1} & = & \exists l,h : low[l] \wedge high[h] \wedge \neg\textbf{adjacent}(h,l) \wedge \\
 & & \textbf{copy}(low) \wedge \textbf{addone}(high) \wedge \textbf{copy}(next) \wedge \\
 & & \textbf{copy-channel} \\
\textbf{a2} & = & \exists l,h,m : low[l] \wedge next[m] \wedge high[h] \wedge \\
 & & \textbf{between}(l,m,h) \wedge \textbf{copy-proc} \\
 & & \wedge\, c_1'[m] \wedge \textbf{allfalse}(c_1) \wedge \\
 & & \textbf{copy-other}(c_1,m) \wedge \textbf{copy}(c_2) \wedge \textbf{copy}(c_3) \\
\textbf{a3} & = & \exists n : c_3[n] \wedge \neg c_3'[n] \wedge next[n] \wedge \\
 & & \textbf{copy}(low) \wedge \textbf{copy}(high) \wedge \textbf{addone}(next) \wedge \\
 & & \textbf{copy}(c_1) \wedge \textbf{copy}(c_2) \wedge \textbf{copy-other}(c_3,n) \\
\textbf{a4} & = & \exists j : c_1[j] \wedge \neg c_1'[j] \wedge \textbf{allfalse}(c_2) \wedge c_2'[j] \wedge \\
 & & \textbf{copy-proc} \wedge \textbf{copy-other}(c_1,j) \wedge \\
 & & \textbf{copy-other}(c_2,j) \wedge \textbf{copy}(c_3) \\
\textbf{a5} & = & \exists j : c_2[j] \wedge \neg c_2'[j] \wedge \textbf{allfalse}(c_3) \wedge c_3'[j] \wedge \\
 & & \textbf{copy-proc} \wedge \textbf{copy}(c_1) \wedge \\
 & & \textbf{copy-other}(c_2,j) \wedge \textbf{copy-other}(c_3,j) \\
\textbf{a6} & = & (\forall j : low'[j] \longleftrightarrow next[j]) \wedge \\
 & & \textbf{copy}(high) \wedge \textbf{copy}(next) \wedge \textbf{copy-channel} \\
\textbf{a} & = & \textbf{a1} \vee \textbf{a2} \vee \textbf{a3} \vee \textbf{a4} \vee \textbf{a5} \vee \textbf{a6} \\
\textbf{sys} & = & \textbf{initial} \wedge \Box(\textbf{a} \vee \textbf{idle}) \\
\textbf{initial} & = & \forall i : (i = 0 \longleftrightarrow low[i]) \wedge (i = 0 \longleftrightarrow high[i]) \wedge \\
 & & (i = 0 \longleftrightarrow next[i]) \wedge \neg c_1[i] \wedge \neg c_2[i] \wedge \neg c_3[i] \\
\textbf{inside-window} & = & \Box\,\forall l,n,h : \\
 & & \left[\begin{array}{rl} & low[l] \wedge next[n] \wedge high[h] \\ \longrightarrow & n = h \vee \textbf{between}(l,n,h) \end{array}\right] \\
\textbf{safety} & = & \textbf{sys} \wedge \neg\textbf{inside-window}
\end{array}
$$

Figure A.15: A Sliding Window Protocol in *LTL*(*MSO*)

values of certain temporal subformulas. The semantics of temporal operators is maintained by constraints on the possible changes of the new configuration variables.

We assume, without loss of generality, that a formula ϕ is in negative normal form, i.e., that negations only occur in front of atomic formulas (as negations can always be "pushed" to the atomic formulas). Note that $\neg(\varphi \,\mathcal{W}\, \psi)$ equals $\neg\psi \,\mathcal{W}\, (\neg\varphi \wedge \neg\psi) \wedge \Diamond\neg\varphi$. Define a *core subformula* of ϕ as a subformula of ϕ which has a temporal operator as its main connective. We will denote by $\psi(i)$ a formula where i is the (possibly) only free variable of ψ. We introduce auxiliary variables to track the values of core subformulas of ϕ, as follows.

- For each core subformula $\psi(i)$ we introduce an auxiliary configuration variable x_ψ. Intuitively, the value of $x_\psi[i]$ represents the same value as $\psi(i)$ at each timepoint.
- For each core subformula of the form $\Diamond\,\psi_1(i)$ we introduce an auxiliary configuration variable $y_{\Diamond\psi_1}$ (called an *eventuality variable*). Intuitively, if the formula $y_{\Diamond\psi_1}[i]$ is true, then the formula $\psi_1(i)$ must be true at some future time point.

The value of any subformula ψ can be represented by an encoding $\langle\langle\psi\rangle\rangle$ into the extended set of configuration variables, together with constraints on the auxiliary variables. We first define the *encoding* $\langle\langle\psi\rangle\rangle$ of a formula ψ as follows. Note that the only change is to replace core subformulas by a corresponding auxiliary variable.

$$
\begin{array}{lcll}
\langle\langle\psi\rangle\rangle & \triangleq & \psi & \text{for } \psi \text{ in MSO} \\
\langle\langle\psi_1 \wedge \psi_2\rangle\rangle & \triangleq & \langle\langle\psi_1\rangle\rangle \wedge \langle\langle\psi_2\rangle\rangle & \\
\langle\langle\psi_1 \vee \psi_2\rangle\rangle & \triangleq & \langle\langle\psi_1\rangle\rangle \vee \langle\langle\psi_2\rangle\rangle & \\
\langle\langle\exists i : \psi_1\rangle\rangle & \triangleq & \exists i : \langle\langle\psi_1\rangle\rangle & \\
\langle\langle\forall i : \psi_1\rangle\rangle & \triangleq & \forall i : \langle\langle\psi_1\rangle\rangle & \\
\langle\langle\exists I : \psi_1\rangle\rangle & \triangleq & \exists I : \langle\langle\psi_1\rangle\rangle & \\
\langle\langle\forall I : \psi_1\rangle\rangle & \triangleq & \forall I : \langle\langle\psi_1\rangle\rangle & \\
\langle\langle\Box\,\psi_1(i)\rangle\rangle & \triangleq & x_{\Box\psi_1}[i] & \\
\langle\langle\Diamond\,\psi_1(i)\rangle\rangle & \triangleq & x_{\Diamond\psi_1}[i] & \\
\langle\langle\psi_1(i) \,\mathcal{W}\, \psi_2(i)\rangle\rangle & \triangleq & x_{\psi_1 \mathcal{W} \psi_2}[i] &
\end{array}
$$

Let **localconstr**(ϕ) be the conjunction of a set of *local constraints* on the auxiliary variables of ϕ as defined below.

1. For each auxiliary variable x_ψ the corresponding local constraint is:

$$\forall i : \Big(x_{\Box\psi_1}[i] \longleftrightarrow \Big[\langle\!\langle \psi_1(i) \rangle\!\rangle \wedge x'_{\Box\psi_1}[i] \Big] \Big) \text{ when } \psi(i) \text{ is } \Box\,\psi_1(i),$$
$$\forall i : \Big(x_{\Diamond\psi_1}[i] \longleftrightarrow \Big[\langle\!\langle \psi_1(i) \rangle\!\rangle \vee x'_{\Diamond\psi_1}[i] \Big] \Big) \text{ when } \psi(i) \text{ is } \Diamond\,\psi_1(i), \text{ and}$$
$$\forall i : \Big(x_{\psi_1 \mathcal{W} \psi_2}[i] \longleftrightarrow \Big[\langle\!\langle \psi_2(i) \rangle\!\rangle \vee \Big(\langle\!\langle \psi_1(i) \rangle\!\rangle \wedge x'_{\psi_1 \mathcal{W} \psi_2}[i] \Big) \Big] \Big)$$
when $\psi(i)$ is $\psi_1(i)\,\mathcal{W}\,\psi_2(i)$.

2. Let $y_{\Diamond\psi_1}, \ldots, y_{\Diamond\psi_k}$ be the set of eventuality variables. We define their local constraint as follows.

$$\bigwedge_{m=1}^{k} \forall i : \big(\big[y_{\Diamond\psi_m}[i] \wedge \neg y'_{\Diamond\psi_m}[i] \big] \longrightarrow \langle\!\langle \psi_m(i) \rangle\!\rangle \big)$$

Intuitively, whenever $y_{\Diamond\psi_m}[i]$ flips from true to false, it has "observed" that $\psi_m(i)$ was true in the previous state. Then we know that $\psi_m(i)$ was true at least once in the past.

We will require that all eventuality variables are false infinitely often and that they become true when appropriate. Let $\mathbf{evconstr}(\phi)$ be the *eventuality constraint*, defined below.

$$\bigwedge_{m=1}^{k} \forall i : \big(\neg y_{\Diamond\psi_m}[i] \wedge \big[y'_{\Diamond\psi_m}[i] \longleftrightarrow x'_{\Diamond\psi_m}[i] \big] \big)$$

Intuitively, that the eventuality variables are false means that they have witnessed the "eventuality" (that which should become true). The second constraint says that they should "reset" — i.e., they should check whether another eventuality should be witnessed, which is the case precisely when $x'_{\Diamond\psi_m}[i]$ is true.

Note that, in case some core subformula ψ does not have a free variable i, the local constraints encode the value of ψ on each of the positions i. This is correct, but perhaps not optimal.

We will transform a formula ϕ into the formula $\langle\!\langle \phi \rangle\!\rangle \wedge \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi)$, which is clearly in Büchi Normal Form. The rest of this section will establish soundness of this transformation — meaning that a formula is satisfiable if and only if the transformed formula is satisfiable. The proof is done in two steps. The first lemma below states properties of the auxiliary variables, while the second proves soundness of the construction.

Lemma A.2 *If* $(M, Val, t) \models \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi)$, *then for all core subformulas* $\psi(i)$ *of* ϕ *we have*

1. $(M, Val, t) \models \forall i : (x_{\Box\psi_1}[i] \longrightarrow \Box\langle\!\langle \psi_1(i) \rangle\!\rangle)$ *for* $\psi(i) = \Box\,\psi_1(i)$
2. $(M, Val, t) \models \forall i : (x_{\Diamond\psi_1}[i] \longrightarrow \Diamond\langle\!\langle \psi_1(i) \rangle\!\rangle)$ *for* $\psi(i) = \Diamond\,\psi_1(i)$
3. $(M, Val, t) \models \forall i : (x_{\psi_1 \mathcal{W} \psi_2}[i] \longrightarrow \langle\!\langle \psi_1(i) \rangle\!\rangle\, \mathcal{W}\, \langle\!\langle \psi_2(i) \rangle\!\rangle)$
   for $\psi(i) = \psi_1(i)\,\mathcal{W}\,\psi_2(i)$.

Proof. 1. Suppose $(M, Val', t) \models x_{\Box\psi_1}[i]$ for some valuation $Val' = Val[i \mapsto m]$. Since $(M, Val, t) \models \Box\,\mathbf{localconstr}(\phi)$ we have

$$(M, Val', t) \models \Box\left(x_{\Box\psi_1}[i] \longleftrightarrow \left[\langle\!\langle \psi_1(i)\rangle\!\rangle \wedge x'_{\Box\psi_1}[i]\right]\right).$$

By induction, it follows that $(M, Val', t') \models \langle\!\langle \psi_1(i)\rangle\!\rangle$ for every $t' \geq t$ and thus $(M, Val', t) \models \Box\,\langle\!\langle \psi_1(i)\rangle\!\rangle$.

2. Suppose $(M, Val', t) \models x_{\Diamond\psi_1}[i]$ for some valuation $Val' = Val[i \mapsto m]$. Suppose that $(M, Val', t) \not\models \Diamond\langle\!\langle \psi_1(i)\rangle\!\rangle$. Then $(M, Val', t) \models \Box\neg\langle\!\langle \psi_1(i)\rangle\!\rangle$.

   Together with

$$(M, Val', t) \models \Box\left(x_{\Diamond\psi_1}[i] \longleftrightarrow \left[\langle\!\langle \psi_1(i)\rangle\!\rangle \vee x'_{\Diamond\psi_1}[i]\right]\right)$$

   from the local constraints, we therefore get $(M, Val', t) \models \Box\, x_{\Diamond\psi_1}[i]$.

   The eventuality constraint gives

$$(M, Val', t') \models y'_{\Diamond\psi_1}[i] \longleftrightarrow x'_{\Diamond\psi_1}[i], \text{ for some } t' \geq t\,.$$

   Then it follows from $(M, Val', t) \models \Box\, x_{\Diamond\psi_1}[i]$ that

$$(M, Val', t') \models y'_{\Diamond\psi_1}[i]$$

   and thus

$$(M, Val',\ t'+1) \models y_{\Diamond\psi_1}[i]\,.$$

   Let $t'' > t'+1$ be the earliest point in time after t' (which has to exist because of the eventuality constraint) when

$$(M, Val', t'') \models \neg y_{\Diamond\psi_1}[i]\,.$$

   But then since t'' was the earliest point in time we have

$$(M, Val',\ t''-1) \models y_{\Diamond\psi_1}[i] \ \wedge\ \neg y'_{\Diamond\psi_1}[i]$$

   which together with the local constraint of $y_{\Diamond\psi}$ gives us

$$(M, Val',\ t''-1) \models \langle\!\langle \psi_1(i)\rangle\!\rangle\,.$$

   Since $t''-1 > t' \geq t$ we conclude that

$$(M, Val', t) \models \Diamond\langle\!\langle \psi_1(i)\rangle\!\rangle$$

   which contradicts the assumption.

3. Suppose $(M, Val', t) \models x_{\psi_1 \mathcal{W} \psi_2}[i]$ for some valuation $Val' = Val[i \mapsto m]$. Since $(M, Val, t) \models \Box\, \mathbf{localconstr}(\phi)$ we have $(M, Val', t) \models$
$\Box \left(x_{\psi_1 \mathcal{W} \psi_2}[i] \longleftrightarrow \left[\langle\!\langle \psi_2(i) \rangle\!\rangle \vee \left(\langle\!\langle \psi_1(i) \rangle\!\rangle \wedge x'_{\psi_1 \mathcal{W} \psi_2}[i] \right) \right] \right)$.

   By induction on t it follows that either $(M, Val', t) \models \Box \langle\!\langle \psi_1(i) \rangle\!\rangle$, or that eventually for some $t' \geq t$ we have $(M, Val', t') \models \langle\!\langle \psi_2(i) \rangle\!\rangle$ before which we have $(M, Val', t'') \models \langle\!\langle \psi_1(i) \rangle\!\rangle$ for each $t'' : t \leq t'' < t'$. Hence $(M, Val', t) \models \langle\!\langle \psi_1(i) \rangle\!\rangle \, \mathcal{W} \, \langle\!\langle \psi_2(i) \rangle\!\rangle$ as desired.

□

Lemma A.3 *Let ψ be a subformula of ϕ, Val a valuation, and t a timepoint. There is a matrix M such that $(M, Val, t) \models \psi$ if and only if there is a matrix M', different from M only in the auxiliary variables of ϕ, such that $(M', Val, t) \models \langle\!\langle \psi \rangle\!\rangle \wedge \Box\, \mathbf{localconstr}(\phi) \wedge \Box\Diamond\, \mathbf{evconstr}(\phi)$.*

Proof. $\Longrightarrow$: Define M' to be the same as M (of width n) except for the auxiliary variables. We will show that the auxiliary variables can be set in M' so that $(M', Val, t) \models \langle\!\langle \psi \rangle\!\rangle \wedge \Box\, \mathbf{localconstr}(\phi) \wedge \Box\Diamond\, \mathbf{evconstr}(\phi)$.

- For each core subformula $\psi'(i)$ of ψ and for each $t'' \in \mathbf{N}$ and $m \in \mathbf{Z}_n$ let:

$$x_{\psi'} \in M'(t'', m) \iff (M, Val[i \mapsto m], t'') \models \psi'(i)\,. \quad (\star)$$

- We show that there exists an infinite sequence of timepoints $(t_k)_{k \geq 0}$ with $t = t_0 < t_1 < \cdots$ such that for each $k > 1$ $(M', Val, t_k) \models \mathbf{evconstr}(\phi)$. For each such t_k and for each core subformula of ψ of the form $\Diamond\, \psi_1(i)$ and $m \in \mathbf{Z}_n$ we thus put:

  - $y_{\Diamond \psi_1} \notin M'(t_k, m)$, and
  - $y_{\Diamond \psi_1} \in M'(1 + t_k, m) \iff x_{\Diamond \psi_1} \in M'(1 + t_k, m)\,. \quad (\star\star)$

  From t_k we find t_{k+1} by defining M' for t' with $t_k < t' \leq t_{k+1}$ inductively, as follows. The strategy we employ is to choose the timepoint t_{k+1} such that the values of each variable $y_{\Diamond \psi_1}$ are all false, i.e.:

  - For each core subformula $\Diamond\, \psi_1(i)$ let

$$y_{\Diamond \psi_1} \in M'(t' + 1, m) \iff y_{\Diamond \psi_1} \in M'(t', m) \wedge (M, Val[i \mapsto m], t') \not\models \psi_1(i)\,.$$

– If for some earliest point in time $t' > t_k$ we for every core subformula $\Diamond\,\psi_1(i)$ and $m \in \mathbf{Z}_n$ have $y_{\Diamond\psi_1} \notin M'(t', m)$ then let $t_{k+1} = t'$.

Thus we allow the values of $y_{\Diamond\psi_1}$ between t_k and t_{k+1} to change from true to false, but not from false to true. Note that the eventuality variables satisfy their local constraints. Now we show that we can always find t_{k+1} from t_k. Suppose, in contrary, that we cannot. Then there is some core subformula $\Diamond\,\psi_1(i)$ and $m \in \mathbf{Z}_n$ such that:

$$y_{\Diamond\psi_1} \in M'(t', m) \text{ for all } \; t' > t_k\,.$$

Since the above holds in particular for $t' = 1 + t_k$ we have by $(\star\star)$ that $x_{\Diamond\psi_1} \in M'(1+t_k, m)$ and therefore by $(\star)$ we get $(M,\, Val[i \mapsto m], 1 + t_k) \models \Diamond\,\psi_1(i)$. Then $(M,\, Val[i \mapsto m], t'') \models \psi_1(i)$ for some $t'' > t_k$ and thus our strategy described above gives $y_{\Diamond\psi_1} \notin M'(t''+1, m)$. This is a contradiction.

$\Longleftarrow$: We prove that $(M',\, Val, t) \models \langle\!\langle\psi\rangle\!\rangle \wedge \Box\mathbf{localconstr}(\phi) \wedge \Box\Diamond\mathbf{evconstr}(\phi)$ implies $(M',\, Val, t) \models \psi$.

Let thus $M = M'$. We proceed by induction over the structure of ψ.

ψ in MSO: Since $\langle\!\langle\psi\rangle\!\rangle = \psi$, we get $(M,\, Val, t) \models \psi$.

$\psi = \psi_1 \vee \psi_2$: We get $(M,\, Val, t) \models \psi_1$ or $(M,\, Val, t) \models \psi_2$ by induction.

$\psi = \psi_1 \wedge \psi_2$: We get $(M,\, Val, t) \models \psi_1$ and $(M,\, Val, t) \models \psi_2$ by induction.

$\psi = \neg\psi_1$: Then ψ must be in MSO, since ψ is in negative normal form.

$\psi = \exists i : \psi_1$: We get $(M,\, Val, t) \models \exists i : \langle\!\langle\psi_1\rangle\!\rangle$ and by the semantics

$$(M,\, Val[i \mapsto m], t) \models \langle\!\langle\psi_1\rangle\!\rangle$$

for some $m \in \mathbf{Z}_n$. Hence $(M,\, Val, t) \models \langle\!\langle\psi_1(m)\rangle\!\rangle$. Since

$$\Box\mathbf{localconstr}(\phi) \wedge \Box\Diamond\mathbf{evconstr}(\phi)$$

is a closed formula, and thus does not depend on i, it follows that

$$(M,\, Val, t) \models \langle\!\langle\psi_1(m)\rangle\!\rangle \wedge \Box\mathbf{localconstr}(\phi) \wedge \Box\Diamond\mathbf{evconstr}(\phi)\,.$$

By the induction hypothesis we get $(M,\, Val, t) \models \psi_1(m)$ and by the semantics we obtain $(M,\, Val, t) \models \exists i : \psi_1(i)$.

$\psi \in \{\exists I : \psi_1, \forall i : \psi_1, \forall I : \psi_1\}$: Analogous with $\psi = \exists i : \psi_1$.

$\psi = \Box\psi_1(i)$: We get $(M, Val, t) \models x_{\Box\psi_1}[i]$. Hence $(M, Val, t) \models \Box\langle\!\langle\psi_1(i)\rangle\!\rangle$ by Lemma A.2. This means that $(M, Val, t') \models \langle\!\langle\psi_1(i)\rangle\!\rangle$ for all $t' \geq t$. By induction we thus obtain $(M, Val, t') \models \psi_1(i)$ for all $t' \geq t$ which means that $(M, Val, t) \models \Box\psi_1(i)$.

$\psi = \Diamond\psi_1(i)$: Analogous with $\psi = \Box\psi_1(i)$.

$\psi = \psi_1(i)\,\mathcal{W}\,\psi_2(i)$: We get $(M, Val, t) \models x_{\psi_1\mathcal{W}\psi_2}[i]$. Hence by Lemma A.2 we have $(M, Val, t) \models \langle\!\langle\psi_1(i)\rangle\!\rangle\,\mathcal{W}\,\langle\!\langle\psi_2(i)\rangle\!\rangle$. By the semantics and the induction hypothesis we thus obtain that either $(M, Val, t) \models \Box\,\psi_1(i)$, or that eventually for some $t' \geq t$ we have $(M, Val, t') \models \psi_2(i)$ before which $(M, Val, t'') \models \psi_1(i)$ for each $t'' : t \leq t'' < t'$. Thus $(M, Val, t) \models \psi_1(i)\,\mathcal{W}\,\psi_2(i)$.

□

We are now ready to prove the main theorem.

Theorem A.4 *For any* restricted formula ϕ *there exists a formula* $BNF(\phi)$ *in Büchi Normal Form such that*

$$M \models \phi \text{ for some matrix } M$$
$$\text{if and only if}$$
$$M' \models BNF(\phi) \text{ for some matrix } M'.$$

Proof. The following formula is in Büchi Normal Form:

$$BNF(\phi) \;=\; \langle\!\langle\phi\rangle\!\rangle \wedge \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi).$$

It follows from Lemma A.3 that there is a matrix M such that $M \models \phi$ if and only if there is a matrix M' such that $M' \models BNF(\phi)$. □

A.8 Verification

As shown in Section A.4.3, to verify that a property holds for a system, we search for models of a formula that is a conjunction of the formula describing the system and the negation of the property. If no such models exist, the property holds. Models of the formula are counterexamples that explain why the property does not hold. Thus, the verification task is to find models of formulas.

To search for models of formulas, we use *Büchi regular transition systems*, defined below. They play the role of Büchi automata in the automata-theoretic approach but for $LTL(MSO)$ instead of LTL. A Büchi regular transition system is an automaton whose states are words and where the transition relation is represented using a regular set. We

say that a length-preserving relation $\mathcal{R}$ on Σ^* is *regular* if the set (of words over $\Sigma \times \Sigma$)

$$(w(1), w'(1))(w(2), w'(2)) \cdots (w(n), w'(n))$$

such that $(w, w') \in \mathcal{T}$ is regular. The transition relation of a BRTS is given by such a regular length-preserving relation, which can also be described by a finite-state transducer — a finite-state automaton over pairs of words.

Definition A.5 ***(Büchi Regular Transition System)***
A Büchi regular transition system (BRTS) *over an alphabet* Σ *is a tuple* $\langle \Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F} \rangle$ *where*

- $\mathcal{I} \subseteq \Sigma^*$ *is a regular set of words over* Σ *called the set of* initial configurations*,*
- $\mathcal{T} \subseteq \Sigma^* \times \Sigma^*$ *is a regular length-preserving relation on words over* Σ*, called the* transition relation*, and*
- $\mathcal{F} \subseteq \Sigma^* \times \Sigma^*$ *is a regular length-preserving relation on words over* Σ*, called the set of* final transitions*.*

An accepting run *of* $\langle \Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F} \rangle$ *is a matrix* M *such that*

- $M(0) \in \mathcal{I}$*,*
- $(M(i), M(i+1)) \in \mathcal{T}$ *for any* $i \geq 0$*, and*
- $(M(i), M(i+1)) \in \mathcal{F}$ *for infinitely many* i*.* □

In the previous section, we showed how to translate a formula in $LTL(MSO)$ into an equivalent formula in Büchi Normal Form. This form is characterized by BRTS.

Theorem A.6 *For every formula* φ *in Büchi Normal Form, there is a Büchi regular transition system* $\langle \Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F} \rangle$ *such that, for every matrix* M*, we have* $M \models \varphi$ *if and only if* M *is an accepting run of* $\langle \Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F} \rangle$*.*

Proof. Let φ be in Büchi Normal Form

$$\phi_I \wedge \Box \phi_T \wedge \Box \Diamond \phi_F$$

and $\langle \Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F} \rangle$ be the BRTS such that for all matrices M:

- $M(0) \in \mathcal{I} \iff M \models \phi_I$ and
- $(M(0), M(1)) \in \mathcal{T} \iff M \models \phi_T$ and
- $(M(0), M(1)) \in \mathcal{F} \iff M \models \phi_F$.

The BRTS $\langle \Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F} \rangle$ exists because ϕ_I, ϕ_T, ϕ_F are formulas in MSO, and thus can be translated into finite-state automata. □

	Safety	Liveness
Token Pass	5.5	16.0
Token Ring	8.4	9.8
Bakery	13.9	44.2
Burns	39.6	
Szymanski	34.3	
Dijkstra	36.4	
Termination Detection	38.0	
Alternating Bit	179.2	
Sliding Window	1687.2	

Table A.1: *Experimental Results*

Just like in the automata-theoretic approach, checking models of a formula thus reduces into checking for accepting runs of a BRTS. Since the transition relation of a BRTS is length-preserving, the existence of an accepting run can be checked by searching for a reachable loop which contains an accepting state. Unlike the automata-theoretic approach, however, the set of states of a BRTS is infinite, requiring new techniques for finding accepting runs.

The procedure we use for finding accepting runs can roughly be described as follows. First, the set of reachable states is computed as $Inv = \mathcal{I} \circ \mathcal{T}^*$. Secondly, loops are found by identifying identical pairs in $(\mathcal{F} \cap \mathcal{T} \cap (Inv \times Inv)) \circ \mathcal{T}^*$. Thus, the problem reduces to computing transitive closures $\mathcal{R} \circ \mathcal{T}^*$ and reachability sets $\mathcal{I} \circ \mathcal{T}^*$.

We have verified safety properties with our tool for regular model checking with techniques for computing transitive closures and reachability sets from [BJNT00, AJNd02], as well as liveness properties for some of the examples. Execution times are given in the table below.

Conclusions Experience has shown [Nil05] that it is efficient to compute e.g. the invariant $\mathcal{I} \circ \mathcal{T}^*$ using so-called *meta actions*, which are (under-approximations of) the transitive closures of program actions. We typically compute the invariant iteratively as the fixed-point $\mathcal{I} \circ (\bigcup_\alpha \alpha^+)$ where $\{\alpha\}$ are the actions of the program. To identify the individual program actions in a setting where the system and negated property is given as *one* formula, we can either *syntactically* identify them from the formula before the translation to BRTS, or *semantically* identify them after the translation. In the pursuit of a uniform model checker, the semantic approach is more appealing, as we then do not have to distinguish between syntactically different, but equivalent, models of actions.

In [JS07], we explored a way to semantically extract a certain class of actions after the translation, with good results. As a general principle, to obtain syntax independence, whenever we have an efficient technique for computing the transitive closure of a class of actions, we should also investigate how these actions can be extracted from the transducer representation of the transition relation.

A.9 Bibliography

[AJN+04] P.A. Abdulla, B. Jonsson, Marcus Nilsson, Julien d'Orso, and M. Saksena. Regular model checking for MSO + LTL. In *Proc.* 16^{th} *Int. Conf. on Computer Aided Verification*, pages 348–360, 2004.

[AJNd02] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002,* 13^{th} *Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.

[AJNd03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Algorithmic improvements in regular model checking. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, 2003.

[BFL] B. Boigelot, J-M. François, and L. Latour. The Liége automata-based symbolic handler (lash). Available at `http://www.montefiore.ulg.ac.be/~boigelot`.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer Verlag, 2000.

[BLS01] K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized networks. *Journal of Universal Computer Science*, 7(2), 2001.

[BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.

[BLW05] A. Bouajjani, A. Legay, and P. Wolper. Handling liveness properties in (ω-)regular model checking. In *Proceedings of the 6th*

International Workshop on Verification of Infinite-State Systems (Infinity'04), volume 138 of *Electronic Notes in Computer Science*, pages 101–115, 2005.

[BSW69] K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmissions over half duplex lines. *Communications of the ACM*, 2(5):260–261, 1969.

[Del00] G. Delzanno. Automatic verification of cache coherence protocols. In Emerson and Sistla, editors, *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Verlag, 2000.

[DFvG83] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed somputations. *Information Processing Letters*, 16(5):217–219, 1983.

[EK00] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *Proc. 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254, 2000.

[EK03] E.A. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Proc. TACAS '03,* 9^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[EN95] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *Proc.* 22^{th} *ACM Symp. on Principles of Programming Languages*, 1995.

[GR97] D. Giammarresi and A. Restivo. Two-dimensional languages. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words, pages 215–267. Springer-Verlag, Berlin, 1997.

[GS92] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of Szymanski's algorithm. In *Proc. TACAS '98,* 4^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 424–438, 1998.

[HJJ+95] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic

in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[HJJ+96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS '95,* 1^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, 1996.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS '00,* 6^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, 2000.

[JS07] Bengt Jonsson and Mayank Saksena. Systematic acceleration in regular model checking. In *Proceedings 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 131–144. Springer Verlag, 2007.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc.* 9^{th} *Int. Conf. on Computer Aided Verification*, volume 1254, pages 424–435, Haifa, Israel, 1997. Springer Verlag.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.

[Lam74] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LPS93] Nancy A. Lynch and Boaz Patt-Shamir. Distrbuted algorithms, lecture notes for 6.852, fall 1992. Technical Report MIT/LCS/RSS-20, MIT, Jan. 1993.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[Nil05] M. Nilsson. *Regular Model Checking*. PhD thesis, Uppsala University, 2005.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc.* 18^{th} *Annual Symp. Foundations of Computer Science*, pages 46–57. IEEE, 31 October–2 November 1977.

[Pnu82] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1982.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer Verlag, 2000.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In Brinskma and Larsen, editors, *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer Verlag, 2002.

[Sis97] A. Prasad Sistla. Parametrized verification of linear networks using automata as invariants. In O. Grumberg, editor, *Proc.* 9^{th} *Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 412–423, Haifa, Israel, 1997. Springer Verlag.

[Szy90] B. K. Szymanski. Mutual exclusion revisited. In *Proc. Fifth Jerusalem Conference on Information Technology*, pages 110–117, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[Tan96] Andrew S. Tannenbaum. *Computer Networks*. Prentice-Hall, 1996.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 133–191. Elsevier, Amsterdam, 1990.

[Var91] Moshe Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1–2):79–98, 1991.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on*

Computer Aided Verification, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

Paper B

Proving Liveness by Backwards Reachability (Extended Version)

The original version was published in LNCS, volume 4137.

Proving Liveness by Backwards Reachability (Extended Version)[1]

Parosh Aziz Abdulla
parosh@it.uu.se
Uppsala University

Bengt Jonsson
bengt@it.uu.se
Uppsala University

Ahmed Rezine
rahmed@it.uu.se
Uppsala University

Mayank Saksena
mayanks@it.uu.se
Uppsala University

Abstract

We present a new method for proving liveness and termination properties for fair concurrent programs, which does not rely on finding a ranking function or on computing the transitive closure of the transition relation. The set of states from which termination or some liveness property is guaranteed is computed by backward reachability analysis. A central technique for handling concurrency is a check for certain commutativity properties. The method is not complete. However, it can be seen as a complement to other methods for proving termination, in that it transforms a termination problem into a simpler one with a larger set of terminated states. We show the usefulness of our method by applying it to existing programs from the literature. We have also implemented it in the framework of Regular Model Checking, and used it to automatically verify non-starvation for parameterized algorithms.

B.1 Introduction

The last decade has witnessed impressive progress in the ability of tools to verify properties of hardware and software systems automatically (e.g., [BMMR01, CGJ+03, Hol97]). The success has to a large extent concerned safety properties, e.g., absence of run-time errors, deadlocks, race conditions, etc. Progress in verification of liveness properties has been less prominent. A major reason is that they are harder to verify

[1]Supported by the Swedish Research Council (http://www.vr.se/).

than safety properties. For finite-state systems and some parameterized systems, safety properties can be verified by computing (some approximation of) the set of reachable states. In enumerative model checkers [Hol97] verifying liveness properties, requires at least a repeated search through the state space. In symbolic model checkers, a natural but more expensive technique is to compute the transitive closure of the transition relation. Multiple fairness requirements can make the situation even more complicated. For general infinite-state systems, the difference between safety and liveness properties is even larger. E.g., for some classes of systems, such as lossy channel systems, checking safety properties is decidable [AJ96b], whereas checking liveness properties is undecidable [AJ96a].

The general approach for proving liveness involves finding auxiliary assertions associated with well-founded ranking functions and helpful directions (e.g., [MP84]). Finding such ranking functions is not easy, and automation requires techniques adapted to specific data domains. Techniques have been developed for programs with integers or reals [BMS05a, BMS05b, BMS05c, CS01, CS02], for functional programs [LJBA01], and for parameterized systems [FPPZ04a, FPPZ04b].

It is well-known that verification of a liveness property can be reduced to the property of *fair termination* of a Büchi automaton — the property is true if the automaton has no fair infinite computations [VW86, Var91]. Equivalently, we can rephrase the problem as follows. Given a cross product of the program with an observer automaton for the negated property, which accepts all counterexamples of the property. The liveness property is then true if all fair computations eventually never visit any accepting state of the cross product. Therefore, we can alternatively prove liveness by showing that all program states eventually (via the cross product) reach a program state from which no computations reach an accepting state (of the cross product). If we can compute the sets of states from which accepting states are never seen, we have reduced liveness to the problem of checking whether a set of states must eventually reach another set of states.

The main technique of software model checking, using finite-state abstractions [CGJ+03] has been difficult to apply when proving liveness properties, since abstractions may introduce spurious loops that do not preserve liveness properties. Podelski and Rybalchenko therefore extended the framework of predicate abstraction to that of *transition predicate abstraction* [PR04, PR05], which involves finding a finite set of suitable transition predicates — essentially over-approximations of actions — and proving well-foundedness of abstract transitions.

In this paper, we present a new method for proving liveness based on reachability analysis. We do not compute the transitive closure of the transition relation, nor do we explicitly construct ranking functions. The

method gives a way to compute program states which must eventually, assuming fairness, reach a given set of terminated states. While general liveness properties can be considered, as argued for above, the most straight-forward application of our method is to verify so-called *response* properties of form $\Box(S \rightarrow \Diamond T)$, where S and T are sets of states [MP90]. This can be done by computing a set of states which are guaranteed to eventually visit T, and checking whether this set includes S.

We obtain a set of *terminating states* for which termination is guaranteed, by computing the set of states that are backward reachable from a set of *terminated states* under a particular transition relation — a so-called *convergence relation*. Computing the set of backward reachable states is computationally easier than finding ranking functions or computing the transitive closure. Thus, in many cases, liveness properties can be established for a class of systems, provided that there is a powerful way to compute sets of backward reachable states. For many classes of parameterized and infinite-state systems, the set of backward reachable states is computable (e.g., [AJ96b, AČJYK00]). For other classes of infinite-state systems, powerful acceleration techniques have been developed that compute or under-approximate the set of reachable states (e.g., [WB98, ACABJ04]). It should be possible to develop equally powerful techniques for backward reachability analysis, and apply them to proving liveness properties.

The use of convergence relations is inspired by termination proofs for simple programs. For a simple deterministic (non-concurrent) program, the set of states from which termination is guaranteed can be calculated as the set of states that are backward reachable from some terminated state. We generalize this observation to develop techniques for using backward reachability analysis to prove termination for general concurrent programs with arbitrary (weak) fairness (a.k.a. justice) requirements; backward reachability analysis should be the only non-trivial computation on the verified program. A central technique for handling concurrency is a new use of commutativity properties between different actions of the program.

Our technique is not complete, in general. It computes an under-approximation of the states from which termination is guaranteed. In case the approximation of the terminating states does not include the states for which we intend to prove termination, there are several ways to increase the power of the method. Continuing the example with a response property $\Box(S \rightarrow \Diamond T)$, this is the situation where we have found that some S' is terminating, but it is not the case that $S \subseteq S'$. Essentially, it then remains to check the "smaller" termination problem, which asks whether $S - S'$ is terminating, when the terminated states are $T \cup S'$.

- We can repeat the backward reachability analysis, letting the computed under-approximation play the role of terminated states. One then exploits the fact that our convergence relation increases when the set of terminated states increases — another reachability analysis will therefore improve the under-approximation.

- We can also apply other techniques (e.g. complete methods based on ranks or transitive closure computation) to prove termination for the remaining states of interest (hopefully, our technique simplified the problem).

- We have developed a complementary technique, suitable for parameterized systems, which also uses backward reachability analysis. It is introduced in Section B.6.

To show the usefulness of our method, we apply it to all examples used by Podelski and Rybalchenko to illustrate their method of transition invariants in [PR04]. We also apply it to the well-known *alternating bit* protocol. This is an example of a lossy channel system, for which liveness properties are undecidable [AJ96a]. Our example shows that backward reachability analysis (which is guaranteed to terminate [AJ96b]) can prove liveness properties for some of these systems, although in general they are undecidable. Finally, we have implemented our techniques in the framework of regular model checking [AJNS04], and proved starvation-freedom automatically for several parameterized mutual exclusion protocols.

Related Work For infinite-state systems, fair termination is typically proven by finding auxiliary assertions associated with well-founded ranking functions and helpful directions (e.g., [MP84, MP91]). Automated construction of such ranking functions is a challenging task, which requires techniques adapted to specific data domains. Recently, significant progress has been achieved for programs that operate on numerical domains, integers or reals [BMS05a, BMS05b, BMS05c, CS01, CS02, Cou05]. Rather few papers present efficient techniques to prove termination for programs that operate on arbitrary data domains. For families of parameterized systems, where each system instance is finite-state, liveness can in principle be proven by computing the transitive closure of the transition relation; this is typically expensive, or difficult to automatize (requiring, e.g., acceleration) [PS00, AJN$^+$04, JS07]. In [JS07] liveness was proven for the parameterized programs considered in this paper, using transitive closure computation. The essential difference is that there the reachable loops are computed using one "heavy" transitive closure computation, while here we compute many sets of reachable

states using backward reachability computations. No single approach is faster on our entire benchmark, as we will see in Section B.6.

Another approach is to develop heuristics to automate the search for rank functions [FPPZ04a, FPPZ04b] and procedures to check the conditions in a general proof rule [MP91] automatically. A third approach has been to find specialized abstractions, e.g., into integers, which work in certain cases [PXZ02].

Podelski and Rybalchenko extend the framework of predicate abstraction to that of *transition predicate abstraction* [PR04, PR05, PPR05, CPR05], in order to verify liveness properties as fair termination. In order to apply their technique, one needs to find suitable transition predicates — a transition predicate is an over-approximation of a program action. The technique also involves checking well-foundedness of abstract transitions. The approach has been proven effective for programs with linear arithmetic, using linear arithmetic predicates, but is difficult to automatize for general programs. Extensions of predicate abstraction techniques for synthesizing ranking functions have also been developed by Balaban, Pnueli, and Zuck [BPZ05].

Our use of commutativity between actions is inspired by the use of commutativity in partial-order techniques to optimize state-space exploration [CGMP99] in finite-state model checking.

Organization of the Paper Section B.2 contains basic definitions, Section B.3 an informal overview of our method, and Section B.4 the formal presentation of the method. In Section B.5, we illustrate the use of our method on examples considered by Podelski and Rybalchenko [PR04], and the alternating bit protocol. In Section B.6, we give experimental results on non-starvation for parameterized mutual exclusion algorithms, and describe our complementary technique, particularly developed for parameterized systems. Section B.7 contains conclusions.

B.2 Preliminaries

Programs We consider fair concurrent programs modeled as transition systems. A program may contain a set of actions with (weak) fairness requirements (a.k.a. justice).

Formally, a *program* $\mathcal{P}$ is a triple $\langle Q, \longrightarrow, \mathcal{A} \rangle$, where

- Q is a set of *states*,
- $\longrightarrow \subseteq Q \times Q$ is a *transition relation* on Q which includes the identity relation (denoted Id),

- $\mathcal{A}$ is a finite or countable set of *fair actions*, each of which is a subset of $\longrightarrow$, and required to be deterministic.

An *action* is any subset of the transition relation. We write $s \longrightarrow s'$ for $(s, s') \in \longrightarrow$. For an action α, we use $s \xrightarrow{\alpha} s'$ to denote $(s, s') \in \alpha$. An action α is *enabled* in a state s if there is some state s' such that $s \xrightarrow{\alpha} s'$. The set of states in which the action α is enabled is denoted $En(\alpha)$. If S is a set of states, then $S \upharpoonright \alpha$ denotes the set of pairs (s, s') of states such that $s \xrightarrow{\alpha} s'$ and $s \in S$. For a set $\mathcal{B}$ of actions, let $S \upharpoonright \mathcal{B}$ denote $\{S \upharpoonright \alpha \mid \alpha \in \mathcal{B}\}$. By the *complement* $\neg\alpha$ of an action, we mean the action $\longrightarrow - \alpha$.

A *computation* of $\mathcal{P}$ from a state $s \in Q$ is an infinite sequence of states $s_0\ s_1\ s_2\ \cdots$ such that (1) $s = s_0$; (2) $s_i \longrightarrow s_{i+1}$ for each $i \geq 0$; and (3) for each fair action $\alpha \in \mathcal{A}$, there are infinitely many $i \geq 0$ where either $s_i \xrightarrow{\alpha} s_{i+1}$ or $s_i \notin En(\alpha)$.

For a set S of states and action α, let $Pre(\alpha, S)$ be the set of states s such that $s \xrightarrow{\alpha} t$ for some $t \in S$. For a set of actions $\mathcal{B}$, let $Pre^\star(\mathcal{B}, S)$ be the union of S and the set of states s such that $s \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} t$ for some $t \in S$ and $\alpha_1, \cdots, \alpha_n \in \mathcal{B}$.

Termination Given a program $\mathcal{P} = \langle Q, \longrightarrow, \mathcal{A} \rangle$. A *terminated* set of states F is a subset of Q which is *stable*, meaning that whenever $s \in F$ and $s \longrightarrow s'$ then $s' \in F$. Define $\diamond F$, called the *terminating states*, as the set of states s such that any computation of $\mathcal{P}$ from s contains a state in F. In other words, $\diamond F$ is the set of states from which termination is guaranteed, in the sense that each computation from s will eventually reach the terminated states F. A set of states $S \subseteq Q$ *terminates (with respect to F)* if $S \subseteq \diamond F$. In this paper we present techniques for computing (an under-approximation of) $\diamond F$.

Remarks The restriction that each fair action be deterministic can sometimes be circumvented by representing a nondeterministic action as a union of several deterministic ones.

That the terminated states should be stable can always be enforced, by restricting the transition relation. For example, to make $S \subseteq Q$ stable, we could use the restricted transition relation $(\neg S \upharpoonright \longrightarrow) \cup Id$.

Our definition of program does not mention initial states. When initial states are given, a typical use of our techniques will be to first compute the set of reachable states (or an over-approximation), and let them be the states of the program as defined above.

B.3 Informal Overview of the Proof Method

In this section, we give an overview of our method for computing (an under-approximation of) the set $\Diamond F$, where F is a terminated set of states of a program $\mathcal{P} = \langle Q, \longrightarrow, \mathcal{A} \rangle$.

The inspiration for our method is the simple observation that when $\mathcal{P}$ is a deterministic program with only one fair action α, then $\Diamond F$ is the set $Pre^\star(\alpha, F)$. Our goal is therefore a technique for proving termination properties, where the only nontrivial computation is a predecessor calculation, i.e., computing $Pre^\star(\mathcal{B}, S)$ for some set of states S and set of actions $\mathcal{B}$.

Our method works by computing a so-called *convergence relation*, here denoted $\hookrightarrow_F$, on the states of $\mathcal{P}$. We define a convergence relation to be a relation with the property that if $s \hookrightarrow_F t$ and $t \in \Diamond F$ then also $s \in \Diamond F$. From this property it follows that $Pre^\star(\hookrightarrow_F, F) \subseteq \Diamond F$ for any convergence relation $\hookrightarrow_F$.

The construction of a convergence relation $\hookrightarrow_F$ depends in general on F. Once we have constructed $\hookrightarrow_F$ we use it to compute $Pre^\star(\hookrightarrow_F, F)$. Since $\hookrightarrow_F$ is intended to be used in a predecessor calculation, it is natural to allow the use of predecessor calculations also in the construction of $\hookrightarrow_F$ itself (but to avoid computations of transitive closures or other more powerful techniques). In Section B.4 we show a technique for how this can be done.

Introductory Example Our main technique for constructing a convergence relation $\hookrightarrow_F$ uses a commutativity argument to infer that it satisfies the required properties. To explain its intuition, consider the following simple program, which models two deterministic processes executing in parallel.

$$\begin{array}{lllll} \alpha_1 : & x := x - 1 & \text{if} & x > 0 \\ \alpha_2 : & y := y - 1 & \text{if} & y > 0 \end{array}$$

Variables x and y assume values in the natural numbers. Each process executes one action repeatedly — for $i = 1, 2$, process i repeatedly executes action α_i. We model the program as $\mathcal{P} = \langle Q, \longrightarrow, \mathcal{A} \rangle$, where

- $Q = \{(a, b) \mid a, b \in \mathbf{N}\}$ representing values of (x, y),
- $\longrightarrow = \alpha_1 \cup \alpha_2 \cup Id$, and
- $\mathcal{A} = \{\alpha_1, \alpha_2\}$, i.e. both actions are fair.

The set F of terminated states is the single state with $x = y = 0$. We want to examine whether every state in Q is terminating.

In this example, our technique computes $\hookrightarrow_F$ as $\alpha_1 \cup \alpha_2$. Our technique implicitly ascertains that $\hookrightarrow_F$ is a convergence relation using a commutativity argument. To understand why α_1 is in $\hookrightarrow_F$, assume that $s \xrightarrow{\alpha_1} t$ and $t \in \Diamond F$. Consider any computation from s — we argue that it must eventually see F. If it goes first to t we are done. Otherwise, the computation starts with a sequence of executions of action α_2. During this sequence, α_1 remains enabled, and so must eventually (by fairness) be executed, leading to some state u. Now observe that since α_1 and α_2 commute, u is also reachable from t. Since $t \in \Diamond F$ we infer, using the fact that $\Diamond F$ is a stable set, that $u \in \Diamond F$ and hence that $s \in \Diamond F$.

We conclude that, for states in $Pre^\star(\hookrightarrow_F, F)$, termination is guaranteed, which here is the set of all states.

The above technique can prove termination for many programs with a regular structure. It is in general incomplete. For programs where the above technique computes a too small under-approximation of $\Diamond F$, we can proceed in the following ways.

- The backward reachability computation can be repeated several times. If one application produces an under-approximation G of $\Diamond F$, the next application of our technique will compute $\Diamond G$ using a convergence relation $\hookrightarrow_G$ that is larger than in the first computation, since it depends on G instead of F. This increases the set of terminating states.

  Let us illustrate a repeated application by changing the guard of α_1 in the above program into $0 < x \leq y \ \vee \ y = 0$. This destroys commutativity between α_1 and α_2 in case $y = x$. However, a first backward reachability computation will produce the set G consisting of states with $0 \leq x \leq 1$ or with $0 \leq y < x$ as an under-approximation of $\Diamond F$. A second backward reachability computation thereafter reveals that all states are in $\Diamond G$, hence also in $\Diamond F$.

- In many cases, the under-approximation of $\Diamond F$ computed by our technique is sufficiently large that other techniques (e.g., complete techniques based on ranks or transitive closure computation) become computationally feasible.

- In particular, we have developed another technique, which also uses backward reachability. This technique is useful for parameterized systems. It is described in detail in Section B.6.

B.4 Proving Termination as Backward Reachability

In this section we formalize the techniques for calculating (an under-approximation of) the set $\Diamond F$ by backward reachability analysis, presented in the previous section. Given a program $\langle Q, \longrightarrow, \mathcal{A} \rangle$. Let F be a set of terminated states.

Definition B.1 *A* convergence relation *for F is a relation $\hookrightarrow_F \subseteq Q \times Q$ such that*

$$s \hookrightarrow_F t \quad \wedge \quad t \in \Diamond F \quad \Longrightarrow \quad s \in \Diamond F \,.$$

□

The point of convergence relations is that if $\hookrightarrow_F$ is a convergence relation for F, then $Pre^\star(\hookrightarrow_F, F) \subseteq \Diamond F$, i.e., we can use a predecessor calculation to prove that termination is guaranteed from a set of states. Larger convergence relations allow to prove termination for larger sets of states. Any number of convergence relations can also be combined into one, since the union of two convergence relations is again a convergence relation. Furthermore, even if we cannot precisely calculate $Pre^\star(\hookrightarrow_F, F)$, any under-approximation of this set is also in $\Diamond F$.

To apply these ideas, we need techniques to compute sufficiently powerful convergence relations. Now we present our main technique, which is based on a commutativity argument. First we define the key concepts.

Definition B.2 *Let α be a deterministic fair action, and let F be a set of states. Define the* left moving states for (α, F)*, denoted $Left(\alpha, F)$, as the set of states s satisfying*

- *whenever there are states s', t' with $t' \notin F$ such that $s \longrightarrow s' \xrightarrow{\alpha} t'$, then there is a state t with $s \xrightarrow{\alpha} t \longrightarrow t'$.* □

Intuitively, if $s \in Left(\alpha, F)$, then α can "move left" of $\longrightarrow$, and still reach the same state. We say that α is *left commutative* or *left moving* at state s. The definition is illustrated in Figure B.1.

$$\begin{array}{lccc} & s & \longrightarrow & s' \\ \forall\, s', t' & \downarrow\!\alpha & & \downarrow\!\alpha \\ & \exists\, t & \longrightarrow & t' \end{array}$$

Figure B.1: Action α is left commutative at state s — i.e., $s \in Left(\alpha, F)$. Note that $t' \notin F$.

Definition B.3 *The* α-helpful states*, denoted* $Helpful(\alpha, F)$*, is the largest set of states* S *with* $S \subseteq F \cup (En(\alpha) \cap Left(\alpha, F))$ *which satisfies*

$$s \in S \ \wedge \ s \longrightarrow t \ \implies \ s \xrightarrow{\alpha} t \ \vee \ t \in F \cup S \, .$$

□

Intuitively, a state is α-helpful if the properties that α is enabled and left moving remain true when any sequence of transitions not in α are taken, unless F is reached. The above concepts can be used to define a convergence relation as follows. Recall that a terminated set of states is stable.

Theorem B.4 *Let* $\alpha \in \mathcal{A}$ *be a fair action of* $\langle Q, \longrightarrow, \mathcal{A} \rangle$ *and* F *be a terminated set of states. Then the relation* $\overset{\alpha}{\hookrightarrow}_F$ *defined by*

$$\overset{\alpha}{\hookrightarrow}_F \ \triangleq \ Helpful(\alpha, F) \upharpoonright \alpha$$

is a convergence relation for F*.*

Proof. Assume that $s \overset{\alpha}{\hookrightarrow}_F t$ and $t \in \Diamond F$. Consider any computation $s_0 \ s_1 \ s_2 \ \cdots$ from $s = s_0$. We must show that it contains a state in F.

- If there is a k with $s_k \in F$ we are done.
- Otherwise, if there is a k with $s_k \xrightarrow{\alpha} s_{k+1}$, let k be the least such index. By induction, using the definition of $Helpful(\alpha, F)$, we infer that for $i = 0, \ldots, k$ $s_i \in Helpful(\alpha, F)$; hence $s_i \in En(\alpha)$ and $s_i \in Left(\alpha, F)$.

  Let t_i be the unique state with $s_i \xrightarrow{\alpha} t_i$. In particular $t_k = s_{k+1}$. By induction we infer, using the definition of $Left(\alpha, F)$, that t_i is reachable from t for all i with $0 \leq i \leq k$. In particular, $t_k = s_{k+1}$ is reachable from t. From $t \in \Diamond F$ and the stability of $\Diamond F$, we infer $s_{k+1} \in \Diamond F$ and hence the computation must contain a state in F. An illustration of this argument is provided in Figure B.2.
- Otherwise, we infer by induction over k, using $s \in Helpful(\alpha, F)$, that α is enabled in all states of the computation. By fairness, α will eventually be executed, and we are back to the previous case.

□

Corollary B.5 $Pre^\star(\{ \overset{\alpha}{\hookrightarrow}_F \mid \alpha \in \mathcal{A}\}, F) \subseteq \Diamond F$. □

In order to show how termination can be proven by backward reachability analysis, we must finally explain how to compute $Helpful(\alpha, F)$,

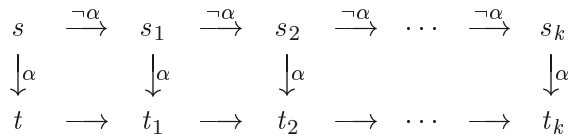


Figure B.2: Illustration of the commutativity principle. $s \overset{\alpha}{\hookrightarrow}_F t$. The α-successor of any $\neg\alpha$-successor of s is either reachable from t, or in F.

or an under-approximation of it, by backward reachability analysis. We first observe that:

$$\mathit{Left}(\alpha, F) = \neg \mathit{Pre}\,((\longrightarrow \circ\, \alpha) - (\alpha \circ \longrightarrow), \neg F)\,.$$

In other words, the set of states $\mathit{Left}(\alpha, F)$ consists of those states from which there is no state in $\neg F$ which can be reached by executing $\longrightarrow$ followed by α, but not by α followed by $\longrightarrow$.

Proposition B.6 *The set* $\mathit{Helpful}(\alpha, F)$ *is the complement of the set*

$$\mathit{Pre}^\star(\neg F \restriction \neg\alpha\,, \neg F \cap (\neg \mathit{Left}(\alpha, F) \cup \neg \mathit{En}(\alpha)))\,.$$

Proof. According to Definition B.3, a state s is in $\mathit{Helpful}(\alpha, F)$ if α remains enabled, and only states in $\mathit{Left}(\alpha, F)$ are visited, unless α is executed or F is seen. It follows that a state t is not in $\mathit{Helpful}(\alpha, F)$ if a state not in F and not in $\mathit{Left}(\alpha, F)$ or $\mathit{En}(\alpha)$ can be reached, without executing α or seeing F. The set of such states t is exactly what is computed above (before taking the complement). □

B.5 Examples

In this section we illustrate our technique, defined in Section B.4, by applying it to examples from the literature. We consider the examples *Any-Down*, *Choice*, *Conc-Whiles* and *Loops* used by Podelski and Rybalchenko to illustrate their method of transition invariants [PR04]. Our technique can handle all the examples given in [PR04] in two iterations or less. We also consider the *alternating bit protocol*.

B.5.1 Any-Down

This is the program *Any-Down* [PR04]. For readability, we reformulate the program into the action-based syntax of the example in Section B.3, as follows.

$$\begin{array}{lcll} \alpha_1 & : & y := y + 1 & \text{if} \quad x = 1 \\ \alpha_2 & : & x := 0 & \text{if} \quad \mathit{true} \\ \alpha_3 & : & y := y - 1 & \text{if} \quad x = 0 \ \wedge \ y > 0 \end{array}$$

The program variable y assumes values in the natural numbers, and the variable x assumes values in $\{0, 1\}$. Both α_2 and α_3 are fair actions. The transition relation is the union of all three actions plus the identity relation. The set F of terminated states is the single state with $x = y = 0$. It is well-known that a standard termination proof for this program will require a ranking function whose range is larger than the natural numbers. This suggests that we need at least two iterations of our technique to compute the set $\Diamond F$, as, intuitively, one iteration covers at most ω computation steps.

In the first iteration we compute $\mathit{Helpful}(\alpha_i, F)$ for $i = 2, 3$. The computations are summarized in the table below.

α_i	$\mathit{En}(\alpha_i)$	$\mathit{Left}(\alpha_i, F)$	$\mathit{Helpful}(\alpha_i, F)$
α_2	true	$x = 0$	$x = 0$
α_3	$x = 0 \ \wedge \ y > 0$	$x = 0 \ \vee \ y = 0 \ \vee \ y = 1$	$x = 0$

We explain the entries of the table for α_2. The set $\mathit{Left}(\alpha_2, F)$ includes all states s where $x = 0$. This is since either (1) $y = 0$ in which case $s \in F$; or (2) $y > 0$, which means that α_1 is not enabled, and α_2 commutes with α_3. On the other hand, $\mathit{Left}(\alpha_2, F)$ does not include any state s with $x = 1$, which we see as follows. Consider $s \xrightarrow{\alpha_1} \xrightarrow{\alpha_2} t$, for some t with $y > 0$. Clearly $t \notin F$, but we do not have $s \xrightarrow{\alpha_2} \longrightarrow t$.

The set $\mathit{Helpful}(\alpha_2, F)$ includes all states s where $x = 0$. The action α_2 stays enabled from such a state s. Furthermore, the action α_1 is disabled, while the execution of α_3 from s again leads to a state in $\mathit{Helpful}(\alpha_2, F)$.

By Corollary B.5, the following set is in $\Diamond F$:

$$G \ \stackrel{\triangle}{=} \ \mathit{Pre}^\star(\{\mathit{Helpful}(\alpha_i, F) \restriction \alpha_i \mid i = 2, 3\}, F) \quad \equiv \quad x = 0$$

In the second iteration we repeat the procedure with a larger set of terminated states G. The interesting difference is that $\mathit{Left}(\alpha_2, G)$, which is true, is larger than $\mathit{Left}(\alpha_2, F)$, since any execution of α_2 leads to G. Hence also $\mathit{Helpful}(\alpha_2, G)$, which is true, is larger than $\mathit{Helpful}(\alpha_2, F)$.

α_i	$\mathit{En}(\alpha_i)$	$\mathit{Left}(\alpha_i, G)$	$\mathit{Helpful}(\alpha_i, G)$
α_2	true	true	true
α_3	$x = 0 \ \wedge \ y > 0$	true	$x = 0$

Again, by Corollary B.5, the following set is in $\Diamond G$, hence in $\Diamond F$:

$$\mathit{Pre}^\star(\{ \overset{\alpha_2}{\hookrightarrow}_G , \overset{\alpha_3}{\hookrightarrow}_G\}, G) \quad \equiv \quad \mathit{true}$$

Hence, all states are terminating. □

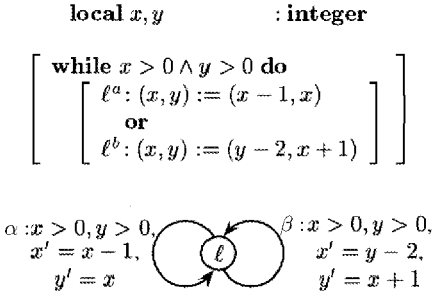


Figure B.3: The program Choice.

B.5.2 Choice

This is the program *Choice* [PR04].

$$\begin{array}{lcll} \alpha & : & (x,y) := (x-1,x) & \text{if} \quad x>0 \ \wedge \ y>0 \\ \beta & : & (x,y) := (y-2,x+1) & \text{if} \quad x>0 \ \wedge \ y>0 \end{array}$$

The actions are taken in a loop while $x > 0 \ \wedge \ y > 0$. The question is whether the loop terminates.

Both actions are fair. A state is of the form (x, y), where x and y are in $\mathbf{Z}$. We consider any initial values where $x > 0 \ \wedge \ y > 0$ (otherwise the loop trivially terminates, as no action is enabled). We let the set S of states be those that are reachable from the initial states. The terminated states are $F \triangleq x = 0 \ \vee \ y = 0$. We obtain the following.

$\cdot$	$En(\cdot) \cap Left(\cdot, F)$	$Helpful(\cdot, F)$
α	$y \leq 2 \ \vee \ x = y-2 \ \vee$ $x = y-1$	$y \leq 2 \ \vee \ x = y-2 \ \vee$ $x = y-1 \ \vee \ F$
β	$x \leq 1$	$x \leq 1 \ \vee \ F$

By Corollary B.5, $G \triangleq Pre^\star(\{\overset{\alpha}{\hookrightarrow}_F, \overset{\beta}{\hookrightarrow}_F\}, F) \subseteq \Diamond F$. We observe that G includes $x \leq 1 \ \vee \ y \leq 2 \ \vee \ x = y-2 \ \vee \ x = y-1$.

In a second iteration, considering G as the set of terminated states, we obtain the sets summarized in the table below.

$\cdot$	$En(\cdot) \cap Left(\cdot, G)$	$Helpful(\cdot, G)$
α	$x > 0 \ \vee \ y > 0$	$x \geq 0 \ \vee \ y \geq 0$
β	$x > 0 \ \vee \ y > 0$	$x \geq 0 \ \vee \ y \geq 0$

Notice that $\overset{\alpha}{\hookrightarrow}_G = \alpha$ since $En(\alpha) \subseteq Helpful(\alpha, G)$. We also have $\overset{\beta}{\hookrightarrow}_G = \beta$.

By Corollary B.5, therefore $Pre^\star(\{\alpha, \beta\}, G) \subseteq \Diamond G \subseteq \Diamond F$, which includes the initial values. Thus the loop always terminates. □

B.5.3 Conc-Whiles

This is the program *Conc-Whiles* [PR04].

$$\begin{array}{rl} \textbf{local } x, y & : \textbf{integer where } x > 0, y > 0 \\ & P_1 :: \left[\begin{array}{l} \ell_0 : \textbf{while } x > 0 \textbf{ do} \\ \quad \ell_1 : y := x - 1 \\ \ell_2 : y := 0 \end{array}\right] \\ \| & P_2 :: \left[\begin{array}{l} m_0 : \textbf{while } y > 0 \textbf{ do} \\ \quad m_1 : x := y - 1 \\ m_2 : x := 0 \end{array}\right] \end{array}$$

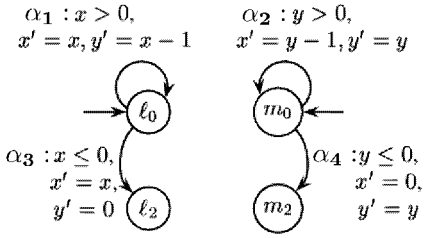


Figure B.4: The program Conc-Whiles.

$$\begin{array}{lcll} \alpha_1 & : & (x, y) := (x, x - 1) & \text{if} \quad l = l_0 \ \wedge \ x > 0 \\ \alpha_2 & : & (x, y) := (y - 1, y) & \text{if} \quad m = m_0 \ \wedge \ y > 0 \\ \alpha_3 & : & (l, x, y) := (l_2, x, 0) & \text{if} \quad l = l_0 \ \wedge \ x \leq 0 \\ \alpha_4 & : & (m, x, y) := (m_2, 0, y) & \text{if} \quad m = m_0 \ \wedge \ y \leq 0 \end{array}$$

This program models two processes running in parallel. All actions are fair. A state is of the form (l, m, x, y), where $l \in \{l_0, l_2\}$, $m \in \{m_0, m_2\}$, and $x, y \in \mathbf{Z}$. Any state where $l = l_0$, $m = m_0$, $x > 0$ and $y > 0$ is an initial state. We let S be the states reachable from the initial states. The terminated states are $F \stackrel{\triangle}{=} l = l_2 \ \wedge \ m = m_2 \ \wedge \ x = 0 \ \wedge \ y = 0$. We obtain the following.

α_i	$En(\alpha_i) \cap Left(\alpha_i, F)$	$Helpful(\alpha_i, F)$
α_1	$\{(l_0, m_0, x, x+1) \mid x > 0\}$ $\cup \{(l_0, m_0, x, 0) \mid x \geq 1\}$	$\{(l_0, m_0, x, x+1) \mid x > 0\} \cup F$
α_2	$\{(l_0, m_0, y+1, y) \mid y > 0\}$ $\cup \{(l_0, m_0, 0, y) \mid y \geq 1\}$	$\{(l_0, m_0, y+1, y) \mid y > 0\} \cup F$
α_3	$\{(l_0, m_0, 0, y) \mid y \geq 0\}$ $\cup \{(l_0, m_2, 0, 0)\}$	$\{(l_0, m_0, 0, y) \mid y = 0, 1\}$ $\cup \{(l_0, m_2, 0, 0)\} \cup F$
α_4	$\{(l_0, m_0, x, 0) \mid x \geq 0\}$ $\cup \{(l_2, m_0, 0, 0)\}$	$\{(l_0, m_0, x, 0) \mid x = 0, 1\}$ $\cup \{(l_2, m_0, 0, 0)\} \cup F$

By Corollary B.5, the set $G \stackrel{\triangle}{=} Pre^\star(\{ \overset{\alpha_i}{\hookrightarrow}_F \mid i = 1..4\}, F) \subseteq \Diamond F$.

Note that G includes $\{(l_0, m_0, x, x+1) \mid x \geq 0\}$, $\{(l_0, m_0, x+1, x) \mid x \geq 0\}$, and $\{(l, m, 0, 0) \mid l = l_0, l_2, m = m_0, m_2\}$, but not all of S. Repeating the procedure on G gives the following.

α_i	$En(\alpha_i) \cap Left(\alpha_i, F)$	$Helpful(\alpha_i, G)$
α_1	$\{(l_0, m_0, x, y) \mid x > 0, y \geq 0\}$	$\{(l_0, m_0, x, y) \mid x > 0\} \cup F$
α_2	$\{(l_0, m_0, x, y) \mid x \geq 0, y > 0\}$	$\{(l_0, m_0, x, y) \mid y > 0\} \cup F$
α_3	$\{(l_0, m_0, 0, y) \mid y \geq 0\}$ $\cup \{(l_0, m_2, 0, 0)\}$	$\{(l_0, m_0, 0, y) \mid y \geq 0\}$ $\cup \{(l_0, m_2, 0, 0)\} \cup F$
α_4	$\{(l_0, m_0, x, 0) \mid x \geq 0\}$ $\cup \{(l_2, m_0, 0, 0)\}$	$\{(l_0, m_0, x, 0) \mid x \geq 0\}$ $\cup \{(l_2, m_0, 0, 0)\} \cup F$

By Corollary B.5, the set $Pre^\star(\{ \overset{\alpha_i}{\hookrightarrow}_G \mid i = 1..4\}, G)$, which includes the initial states, is in $\Diamond G$, and therefore in $\Diamond F$. □

B.5.4 Loops

This is the program *Loops* [PR04].

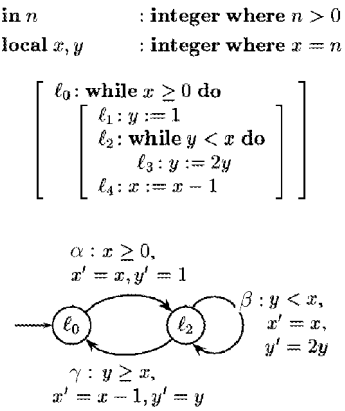


Figure B.5: The program Loops.

$$
\begin{array}{lcll}
\alpha & : & (x, y) := (x, x-1) & \text{if} \quad l = l_0 \ \wedge \ x > 0 \\
\beta & : & (x, y) := (y-1, y) & \text{if} \quad m = m_0 \ \wedge \ y > 0 \\
\gamma & : & (l, x, y) := (l_2, x, 0) & \text{if} \quad l = l_0 \ \wedge \ x \leq 0
\end{array}
$$

In this program, all actions are fair. A state is of the form (l, x, y), where $l \in \{l_0, l_2\}$ and $x, y \in \mathbf{Z}$. Any state (l_0, x, y) with $x > 0$ is an initial state. Let the set of terminated states F be $\{(l_0, x, y) \mid x < 0\}$. We wonder whether all initial states terminate.

$\cdot$	$En(\cdot) \cap Left(\cdot, F)$	$Helpful(\cdot, F)$
α	$\{(l_0, x, y) \mid x \geq 0\}$	$\{(l_0, x, y) \mid x \geq 0\} \cup F$
β	$\{(l_2, x, y) \mid y < x\}$	$\{(l_2, x, y) \mid y < x\} \cup F$
γ	$\{(l_2, x, y) \mid x \leq y\}$	$\{(l_2, x, y) \mid x \leq y\} \cup F$

Note that $\overset{\alpha}{\hookrightarrow}_F = \alpha$ since $En(\alpha) \subseteq Helpful(\alpha, F)$. The corresponding holds for β and γ. By Corollary B.5 therefore $Pre^\star(\{\alpha, \beta, \gamma\}, F)$, which includes the initial states, is in $\Diamond F$. □

B.5.5 Alternating Bit Protocol

This protocol consists of finite-state processes that communicate over unbounded and lossy FIFO channels. As shown in our earlier work, it is decidable whether such a protocol satisfies a safety property [AJ96b], but undecidable whether a protocol satisfies a liveness property [AJ96a]. Using our technique, we can prove liveness properties for some of these protocols.

The alternating bit protocol involves a *sender* and a *receiver* that communicate over two channels c_M and c_A. Channel c_M is used to transmit messages from the *sender* to the *receiver*, and channel c_A to transmit acknowledgments from the *receiver* to the *sender*. Both channels are FIFO and faulty in the sense that messages may be lost but not reordered. The purpose of the protocol is to transmit messages from the *sender* to the *receiver* in correct order, in spite of the fact that the channels can lose messages. Corruption of messages can also be taken into account by modeling it as a loss (some mechanism will detect and discard a corrupted message). Each channel is "fair" in the sense that if infinitely many messages are input, then infinitely many messages will be delivered.

We describe the operations of *sender* and *receiver* in the protocol. At one end of the channels, the *sender* constructs a message m_b by adding a sequence number b in $\{0, 1\}$ to a pending message m, and sends it on the channel c_M to the *receiver*. The *sender* waits for an acknowledgment a_b with the same sequence number on the channel c_A. If a_b arrives, the procedure is repeated with the next pending message but with an inverted sequence number $(1 - b)$. If no acknowledgment a_b arrives within a specified time period the *sender* retransmits the message m_b. Retransmissions are repeated until an acknowledgment a_b with a corresponding sequence number arrives. Acknowledgments with non-corresponding sequence numbers are discarded. On the other end of the channels, the *receiver* receives messages m_b from the incoming channel c_M. A message m_b is delivered if the corresponding sequence number b was expected. After delivery of m_b, the *receiver* sends on channel c_A an acknowledg-

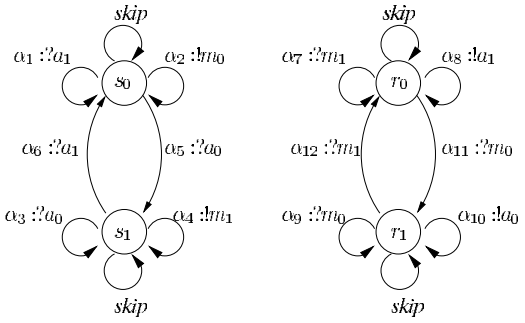


Figure B.6: The alternating bit protocol.

ment a_b with the same sequence number to the *sender*. The *receiver* expects a message with an inverted sequence number $(1 - b)$. Messages with non-expected sequence numbers are discarded.

The *sender* and the *receiver* are modeled by the finite-state processes depicted in Figure B.6. The states of the *sender* are in $\{s_0, s_1\}$, while those of the *receiver* are in $\{r_0, r_1\}$. A state of the system is of form $s\, r\, (w_M, w_A)$ where s is a sender state, r is a receiver state, w_M is the content of channel c_M, and w_A is the content of channel c_A. The initial state is $s_0 r_0(\langle\rangle, \langle\rangle)$ with both channels empty. From a state $s\, r\, (w_M, w_A)$, the effects of the actions $!m_b$ and $!a_b$, with b in $\{0, 1\}$, are respectively $s\, r\, (m_b \cdot w_M, w_A)$ and $s\, r\, (w_M, a_b \cdot w_A)$ — the operator "$\cdot$" is concatenation of channel content. The state $s\, r\, (w_M, w_A)$ results from applying the action $?m_b$ to the state $s\, r\, (w_M \cdot m_b, w_A)$, or from applying the action $?a_b$ to the state $s\, r\, (w_M, w_A \cdot a_b)$. Here, c_M and c_A are perfect FIFO buffers, and message losses are modeled as a non-deterministic choice between a send and a *skip* action.

There are techniques to automatically calculate the set of states reachable from the initial state $s_0 r_0(\langle\rangle, \langle\rangle)$. An example is to start from the initial state and to apply the technique of *loop-first search* [BG96]. This technique generates the set of reachable states by taking all possible transitions, and evaluating (whenever possible) the effect of performing an arbitrary number of the same transition. Examples of such *accelerations* are α_1^* or α_2^*, resulting in the addition of to the tail of c_M, respectively the substraction from the head of c_A, of an arbitrary number of m_0, respectively a_1. Sets of states can be captured by *Queue-content Decision Diagram (QDD)* of the form $s\, r\, (w_M, w_A)$ where w_M and w_A are regular languages. The search stops once the set of generated states stabilizes, i.e. no new states are generated when applying transitions. For the protocol at hand, this technique returns the set of reachable states as the union of the four sets $s_0 r_0(m_0^* m_1^*, a_1^*)$, $s_0 r_1(m_0^*, a_0^* a_1^*)$, $s_1 r_0(m_1^*, a_1^* a_0^*)$, and $s_1 r_1(m_1^* m_0^*, a_0^*)$.

We describe the program $\langle Q, \longrightarrow, \mathcal{A}\rangle$ corresponding to the *alternating bit protocol*. The set Q is here chosen to be the set of reachable states computed above. The transition relation $\longrightarrow$ is the union of the actions *skip* and $\alpha_1, \ldots, \alpha_{12}$. All these actions, except *skip*, are in $\mathcal{A}$. This corresponds to the assumption that if a message is continuously retransmitted, then eventually one of the messages is not lost.

We use the technique defined in Section B.4 to prove the following four progress properties of the protocol.

$$
\begin{array}{lcll}
P_{r_0r_1} & : & s_0r_0(m_0^*m_1^*, a_1^*) & \subseteq \ \Diamond s_0r_1(m_0^*, a_0^*a_1^*) \\
P_{s_0s_1} & : & s_0r_1(m_0^*, a_0^*a_1^*) & \subseteq \ \Diamond s_1r_1(m_1^*m_0^*, a_0^*) \\
P_{r_1r_0} & : & s_1r_1(m_1^*m_0^*, a_0^*) & \subseteq \ \Diamond s_1r_0(m_1^*, a_1^*a_0^*) \\
P_{s_1s_0} & : & s_1r_0(m_1^*, a_1^*a_0^*) & \subseteq \ \Diamond s_0r_0(m_0^*m_1^*, a_1^*)
\end{array}
$$

Observe that property $P_{r_0r_1}$ implies that from any state in $s_0r_0(m_0^*m_1^*, a_1^*)$, the system is guaranteed to reach a state in $s_0r_1(m_0^*, a_0^*a_1^*)$. This means the *receiver* changed state from r_0 to r_1. In other words, the receiver is guaranteed to take action α_{11} and to receive the message m_0. A similar reasoning with $P_{s_0s_1}$, $P_{r_1r_0}$ and $P_{s_1s_0}$ ensures *sender* and *receiver* indefinitely alternate sending m_0, a_0, m_1 and a_1. We show in the following how to prove the property $P_{r_0r_1}$; the other properties can be proven similarly.

Let $F \stackrel{\triangle}{=} s_0r_1(m_0^*, a_0^*a_1^*)$. We use the technique defined in Section B.4 to calculate a set of states included in $\Diamond F$. To ensure the stability of F, we first modify all actions α to $\neg F \upharpoonright \alpha$. The sets where the actions are enabled are shown in Figure B.7. Observe that $\neg F \upharpoonright \alpha_5$ is empty. The results of the computations (according to Proposition B.6) of the helpful set of states for each fair action α_i in $\mathcal{A}$ appear in the same figure. Let us give an intuition of why $\mathit{Helpful}(\alpha_7, F)$ equals the union of the sets $s_0r_0(m_0^*m_1^+, a_1^*)$, $s_1r_0(m_1^+, a_1^*a_0^*)$ and $s_0r_1(m_0^*, a_0^*a_1^*)$. For every state s in this union, it is the case that either (1) s is in $F = s_0r_1(m_0^*, a_0^*a_1^*)$; or (2) s is in the union of $s_0r_0(m_0^*m_1^+, a_1^*)$ and $s_1r_0(m_1^+, a_1^*a_0^*)$. In the second case, observe that α_7 is enabled and that the only action that does not commute with α_7 is action α_{11} (which is not enabled). We have that (1) α_7 is enabled from s and commutes with any other enabled action; and (2) the execution of any other action from s leads to the same union of $s_0r_0(m_0^*m_1^+, a_1^*)$ and $s_1r_0(m_1^+, a_1^*a_0^*)$. Observe that α_7 is not enabled outside the union of $s_0r_0(m_0^*m_1^+, a_1^*)$, $s_1r_0(m_1^+, a_1^*a_0^*)$ and $s_0r_1(m_0^*, a_0^*a_1^*)$.

By Corollary B.5, we have $G \stackrel{\triangle}{=} \mathit{Pre}^\star(\{ \stackrel{\alpha_i}{\hookrightarrow}_F \mid i = 1..12\}, F) \subseteq \Diamond F$. Observe that $s_0r_0(m_0^*m_1^*, a_1^*) = \mathit{Pre}^\star(\{ \stackrel{\alpha_i}{\hookrightarrow}_F \mid i = 2, 7, 11\}, s_0r_0(m_0^*, a_1^*)) \subseteq G$. Hence $s_0r_0(m_0^*m_1^*, a_1^*) \subseteq \Diamond F$. □

α_i	$En(\alpha_i)$	$En(\alpha_i) \cap Left(\alpha_i, F)$	$Helpful(\alpha_i, F)$
α_1	$s_0r_0(m_0^*m_1^*, a_1^+)$	$s_0r_0(m_0^*m_1^*, a_1^+)$	$s_0r_0(m_0^*m_1^*, a_1^+) \cup$ $s_0r_1(m_0^*, a_0^*a_1^*)$
α_2	$s_0r_0(m_0^*m_1^*, a_1^*)$	$s_0r_0(m_0^*m_1^*, a_1^*)$	$s_0r_0(m_0^*m_1^*, a_1^*) \cup$ $s_0r_1(m_0^*, a_0^*a_1^*)$
α_3	$s_1r_0(m_1^*, a_1^*a_0^+)$ $\cup s_1r_1(m_1^*m_0^*, a_0^+)$	$s_1r_0(m_1^*, a_1^*a_0^+)$ $\cup s_1r_1(m_1^*m_0^*, a_0^+)$	$s_1r_0(m_1^*, a_1^*a_0^+) \cup$ $s_1r_1(m_1^*m_0^*, a_0^+)$ $\cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_4	$s_1r_0(m_1^*, a_1^*a_0^*)$ $\cup s_1r_1(m_1^*m_0^*, a_0^*)$	$s_1r_0(m_1^*, a_1^*a_0^*)$ $\cup s_1r_1(m_1^*m_0^*, a_0^*)$	$s_1r_1(m_1^*m_0^*, a_0^*) \cup$ $s_0r_1(m_0^*, a_0^*a_1^*)$
α_5	$\emptyset$	$\emptyset$	$s_0r_1(m_0^*, a_0^*a_1^*)$
α_6	$s_1r_0(m_1^*, a_1^+)$	$\emptyset$	$s_0r_1(m_0^*, a_0^*a_1^*)$
α_7	$s_0r_0(m_0^*m_1^+, a_1^*)$ $\cup s_1r_0(m_1^+, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^+, a_1^*)$ $\cup s_1r_0(m_1^+, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^+, a_1^*) \cup$ $s_1r_0(m_1^+, a_1^*a_0^*)$ $\cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_8	$s_0r_0(m_0^*m_1^*, a_1^*)$ $\cup s_1r_0(m_1^*, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^*, a_1^*)$ $\cup s_1r_0(m_1^*, a_1^*a_0^*)$	$s_0r_0(m_0^*m_1^*, a_1^*) \cup$ $s_1r_0(m_1^*, a_1^*a_0^*)$ $\cup s_0r_1(m_0^*, a_0^*a_1^*)$
α_9	$s_1r_1(m_1^*m_0^+, a_0^*)$	$s_1r_1(m_1^*m_0^+, a_0^*)$	$s_1r_1(m_1^*m_0^+, a_0^*) \cup$ $s_0r_1(m_0^*, a_0^*a_1^*)$
α_{10}	$s_1r_1(m_1^*m_0^*, a_0^*)$	$s_1r_1(m_1^*m_0^*, a_0^*)$	$s_0r_1(m_0^*, a_0^*a_1^*)$
α_{11}	$s_0r_0(m_0^+, a_1^*)$	$s_0r_0(m_0^+, a_1^*)$	$s_0r_0(m_0^+, a_1^*) \cup$ $s_0r_1(m_0^*, a_0^*a_1^*)$
α_{12}	$s_1r_1(m_1^+, a_0^*)$	$\emptyset$	$s_0r_1(m_0^*, a_0^*a_1^*)$

Figure B.7: Alternating bit protocol. Calculation of $\Diamond\, s_0r_1(m_0^*, a_0^*a_1^*)$

B.6 Parameterized Systems

In this section we consider verification of liveness properties for parameterized systems; these are systems with an arbitrary number of similar (finite-state) processes operating in parallel. Although each instance of a parameterized system is finite, the union of all instances of the system is infinite-state, since the number of processes is unbounded. Therefore the task of verifying a property for all instances of the system, so-called *uniform verification*, is an infinite-state verification problem. We describe an implementation of our method in the framework of Regular Model Checking [AJNS04]. For several examples in this section, the technique of Section B.4 computes a strict under-approximation of the set $\Diamond F$; therefore we also present a complementary technique which can prove termination for those examples.

Example: Szymanski's Algorithm As an example of a parameterized system, we describe the mutual exclusion algorithm by Szymanski [Szy90]. In the algorithm, an arbitrary number of processes compete for a critical section. The processes are numbered, say from 1 to N.

The *local state* of each process consists of a control state ranging over the integers from 1 to 7 and of two Boolean flags, w and s. A *global state*, representing the system state, is essentially an array of local states, indexed by the process indices. A pseudo-code description of the behavior of process number i is shown in Figure B.8.

```
1:  await ∀j : j ≠ i : ¬s[j]
2:  w[i], s[i] := true, true
3:  if ∃j : j ≠ i : pc[j] ≠ 1 ∧ ¬w[j]
        then s[i] := false ; goto 4
        else w[i] := false ; goto 5
4:  await ∃j : j ≠ i : s[j] ∧ ¬w[j]
        then w[i], s[i] := false, true
5:  await ∀j : j ≠ i : ¬w[j]
6:  await ∀j : j < i : ¬s[j]
7:  s[i] := false ; goto 1
```

Figure B.8: Szymanski's algorithm. Pseudo code for process i.

For instance, according to the code on line 6, if the control state of a process i is 6, and if the value of s is *false* for all processes $j < i$, then the control state of i may be changed to 7. Line 7 represents the critical section. Each process i has an action $\alpha_j(i)$ corresponding to the statement beginning at line j in the pseudo-code for process i. All

actions are fair, except $\alpha_1(i)$; this action represents process i entering the competition for the critical section, and therefore its execution should not be enforced.

Intuitively, and figuratively, the algorithm simulates a number of processes "gathering in a waiting room", after which some process in the waiting room can "shut the door" (i.e., set their s to true) so that no more processes can "enter" (as this disables transitions from line 1), after which the processes "exit the room" in order. Thinking of w and s as "wait" and "shut" may help. Entering the room corresponds to reaching line 2, and "locking the door" to reaching line 5. The difference between "shut" and "lock" is that a shut door can be reopened (the transition from line 3 and line 4 again opens the door), while a locked door remains closed until the processes have exited the room.

Let us explain how mutual exclusion is achieved. Initially, all processes are at line 1 with local variables $w = s = \mathit{false}$. A process executing α_3 will go to line 4 if it sees another process at line 2 and there wait for another process to reach line 5 — otherwise it goes directly to line 5. In the latter case, no-one else entered the room and the process can proceed alone, after locking the door. In the former case, the processes "in the room" eventually meet up at line 5, after which they enter the critical section and exit the room in order.

Starvation freedom can be formulated as follows: whenever any process is at line 2 it will eventually reach line 7. To formalize this property, pick a process k. Define F_k to be all states in which process k is at line 7. To prove starvation freedom for a process k we must show that all reachable states where process k is at line 2 are in $\Diamond F_k$.

B.6.1 A Complementary Termination Rule

In this section, we present a proof rule for termination, which is particularly suitable for the class of parameterized systems considered in this section. It will be used to complement the commutativity-based technique of Corollary B.5 (in Section B.4). The rule assumes that we select a finite number of fair actions of the program, and establishes that a state s is in $\Diamond F$ provided that computations from s satisfy:

- whenever one of the selected actions is enabled, it remains enabled until it is executed,
- each of the actions can be executed at most once before F is reached, and
- when all these actions are disabled, the computation has reached F.

This rule is particularly useful for parameterized systems, since termination is often achieved by letting a selected subset of the processes execute a fixed sequence of actions (i.e., statements). Typically, we will select one parameterized action $\alpha(i)$ for all i, or all i except a distinguished process k for which we want to prove termination. The rule is useful when the actions disable themselves immediately after being executed, which is the case in protocols where actions "change line". The intuition behind the rule, in this case, is that we are able to deduce that $Pre^\star(\{\alpha(i) \mid i \in \mathbf{N}\}, F)$ terminates, if we know that states where all $\alpha(i)$ are disabled (for each i) are in F.

Let us define the involved properties formally. Given a program $\langle Q, \longrightarrow, \mathcal{A}\rangle$. Let F be a terminated set of states.

- $Persist(\alpha, F)$ is the set of states s such that in any computation from s, whenever α is enabled, it remains enabled unless it is executed or F is reached.
- $Twice(\alpha, F)$ is the set of states, from which there exists a computation where α is executed twice (or more) without visiting F.
- Let $\mathcal{B}$ be a finite set of actions. $After(\mathcal{B}, F)$ is the set of states s such that in any computation from s, whenever all actions in $\mathcal{B}$ are disabled at a state s', then $s' \in F$.

Note that the set $\mathcal{B}$ used in $After(\mathcal{B}, F)$ is typically a parameterized set of actions, containing actions of form $\alpha_j(i)$ for a finite set $\{j\}$, and an arbitrary i with $1 \leq i \leq N$. Thus the set $\mathcal{B}$ is unboundedly large. Care must be taken to handle the parameters correctly when performing the predecessor calculations. Now we state the termination rule.

Theorem B.7 *Let $\mathcal{B}$ be a set of fair actions of $\langle Q, \longrightarrow, \mathcal{A}\rangle$, and let F be a set of states in Q. Then*

$$\left[After(\mathcal{B}, F) \cap \bigcap_{\alpha \in \mathcal{B}} [\neg Twice(\alpha, F) \cap Persist(\alpha, F)] \right] \subseteq \Diamond F\,.$$

Proof. Let s be a state in the set defined by the left-hand side. Consider any computation from s. Since $s \in \neg Twice(\alpha, F) \cap Persist(\alpha, F)$ for each $\alpha \in \mathcal{B}$, all actions in $\mathcal{B}$ are enabled until executed or F has been seen, and they cannot be executed twice without seeing F. They might also be disabled at s. Since $s \in After(\mathcal{B}, F)$, when all actions of $\mathcal{B}$ are disabled, we have reached F. By fairness, the actions will eventually be executed, and so we must eventually see F. Thus $s \in \Diamond F$. □

Corollary B.8 *The statement of Theorem B.7 is true also for any under-approximations of the sets $After(\mathcal{B}, F)$, $\neg Twice(\alpha, F)$ and $Persist(\alpha, F)$.* □

The sets used in the rule can be computed using backward reachability analysis, in a manner analogous to the way $\mathit{Helpful}(\alpha, F)$ is computed (see Proposition B.6). We give the computations without proof, as they are rather self-explanatory. Figure B.9 illustrates the computation of $\mathit{Persist}(\alpha, F)$ according to Proposition B.9 — the other computations are similar.

Proposition B.9 *The set $\mathit{Persist}(\alpha, F)$ is the complement of the set*

$$\mathit{Pre}^\star(\neg F \restriction \neg\alpha\,, \neg F \cap \neg \mathit{En}(\alpha))\,.$$

□

Proposition B.10 *The set $\mathit{Twice}(\alpha, F)$ can be computed as follows (in order):*

(1) $S := \mathit{Pre}^\star(\neg F \restriction \longrightarrow, \neg F)$ (2) $S := \mathit{Pre}\,(\neg F \restriction \alpha\,, S)$

(3) $S := \mathit{Pre}^\star(\neg F \restriction \longrightarrow, S)$ (4) $S := \mathit{Pre}\,(\neg F \restriction \alpha\,, S)$

(5) $S := \mathit{Pre}^\star(\neg F \restriction \longrightarrow, S)\,.$

□

Proposition B.11 *The set $\mathit{After}(\mathcal{B}, F)$ is the complement of the set*

$$\mathit{Pre}^\star(\neg F \restriction \neg\alpha\,,\ \neg F\ \cap\ \forall \alpha \in \mathcal{B}\ \neg \mathit{En}(\alpha)\ )\,.$$

□

Figure B.9: Computation of $\mathit{Persist}(\alpha, F)$. In the figure, the state s is in the complement of $\mathit{Persist}(\alpha, F)$.

Optimizations While it is possible to compute the sets $\mathit{Persist}(\alpha, F)$, $\mathit{Twice}(\alpha, F)$ and $\mathit{After}(\mathcal{B}, F)$ as described in Propositions B.9–B.11 — that is how we did in the experiments in [AJRS06] — we will here elaborate on how they can be computed faster. We give sufficient conditions for when reachability computations can be avoided, to save time. We illustrate the conditions using "transition diagrams" of the form

$$S \xrightarrow{\alpha} T$$

where S and T are sets of states. The interpretation of this diagram is: "is it possible to execute α from some state in S and thereby reach some state in T ?"

The mentioned optimizations are sufficiently strong to speed up the application of Theorem B.7 in, e.g., the verification of Szymanski's and Dijkstra's algorithms (see Section B.6.2).

Persist(α, F). If the action α cannot be disabled by other actions (without reaching F), then *Persist*(α, F) is the set of all states. A sufficient check for this is whether

$$En(\alpha) \xrightarrow{\neg\alpha} \neg En(\alpha) \cap \neg F$$

which can be checked without reachability.

Twice(α, F). The intuition here is that we want to ascertain that an action cannot be executed several times (in particular, cannot loop) outside of F. Any sufficient condition for this would work. For example, if the action cannot again be enabled outside of F, i.e.,

$$\neg En(\alpha) \xrightarrow{\neg\alpha} En(\alpha) \cap \neg F$$

then we know it cannot be twice executed before F.

A stronger version, which requires two reachability computations instead of the three in Proposition B.10, is to first compute the set *MayEnable*(α, F) of states s satisfying

$$\underbrace{s \longrightarrow \cdots \longrightarrow}_{\neg F} En(\alpha) \cap \neg F\,.$$

Using a second reachability computation we obtain the set S of states s satisfying

$$\underbrace{s \longrightarrow \cdots \longrightarrow}_{\neg F} \xrightarrow{\alpha} \mathit{MayEnable}(\alpha, F)\,.$$

Clearly α cannot be executed twice (outside of F) in any computation starting in the complement of S. Therefore $\neg S \subseteq \neg\, \mathit{Twice}(\alpha, F)$.

After$(\mathcal{B}, F)$. Without using reachability, we can find out whether *After*$(\mathcal{B}, F)$ is the set of all states, by checking whether

$$[\forall \alpha \in \mathcal{B}\ \neg En(\alpha)] \subseteq F$$

which becomes stronger if we restrict the left hand side to the set of reachable states.

B.6.2 Implementation

We have implemented a verification method based on Corollary B.5 and Theorem B.7 in the framework of Regular Model Checking [AJNS04], and applied it to a number of well-known parameterized mutual exclusion protocols. We have verified the liveness property *individual starvation freedom* — is it always the case that when a process is in a certain local state, it will eventually visit the critical section? More precisely, the properties are of the following form (using pseudo temporal logic):

$$\forall i\ \Box\,(\text{process } i \text{ at local state } x \implies \Diamond(\text{process } i \text{ in critical section}))\,.$$

Verification Procedure For each protocol, we have modeled F_k as the set of states where process k is in the critical section. Note that the property is checked for all possible values of k — i.e. k can be any process. We have thereafter computed an under-approximation G_k of $\Diamond F_k$ using the technique of Section B.4, and thereafter applied the complementary rule described in Section B.6.1 to compute (an under-approximation of) $\Diamond G_k$. To ensure that predecessors are reachable states, we first computed the set of (forwards) reachable states, and restricted all actions to this set.

In our experiments we manually chose what rules to apply and when, to be able to observe their power more precisely. In other words, the experiments were made semi-automatically, in that we gave instructions of the form "first apply this rule a number of times, and then this". For the application of the complementary rule to our examples, we chose the set $\mathcal{B}$ as three types of parameterized actions: $\mathcal{B} = \{\alpha(j) \mid j \in \mathbf{N}\}$, $\mathcal{B} = \{\alpha(k)\}$, or $\mathcal{B} = \{\alpha(j) \mid j \neq k\}$, where α ranges over the actions of the program. The approach can be fully automated by e.g. applying the rules alternatingly. By using sufficient conditions and heuristics, we can try to avoid wasting time on "bad choices" of applications (although any application is sound).

Example: Szymanski's Algorithm We describe how our verification techniques worked when verifying starvation freedom for Szymanski's algorithm. Recall that the pseudo code for this algorithm is given in Figure B.8.

Three successive applications of Corollary B.5 establish starvation freedom for "almost all" the system states where process k is waiting. These are the sets of states of form $Inv \cap k@\{5, 6, 7\}$ — meaning, all reachable states where process k is at lines 5, 6, or 7, for all possible values of k.

However, Corollary B.5 cannot prove starvation freedom for system states where there are processes at both line 1 and line 2. In more detail, we are able to prove that sets of states of form $G = \{1, 3, 4\}^* \cdot k@3 \cdot \{1, 3, 4\}^*$ and $G' = \{2, 3, 4\}^* \cdot k@2 \cdot \{2, 3, 4\}^*$ are in $\Diamond F_k$, where we have used regular expressions over line numbers to describe sets of states. For example, G is the set of all states where an arbitrary number of proceses on lines 1, 3 or 4 occur to the left and right of process k, which itself is on line 3.

Corollary B.5 does not allow us to conclude that $G'' = \{1, 2, 3, 4\}^* \cdot k@2 \cdot \{1, 2, 3, 4\}^*$ is terminating. The reason is that the actions of line 2 may disable the actions of line 1 — i.e., they do not commute. It is however the case that the actions of line 2 are persistent, and cannot be executed twice before F, and the set of states where no processes are at line 2 is known to be in $\Diamond F$, so that one application of Theorem B.7

Model	Time	Starvation Free States
Token Pass	9 s	Whenever the token is to the left of k
Token Ring	14 s	Whenever process k does not have the token
Bakery	36 s	Whenever process k has taken a ticket
Szymanski*	5 min 45 s	Whenever k is at line 2
Burns	7 min 30 s	All processes from line 5, and the first process from any line
Dijkstra*	38 min 43 s	Whenever $p = k$, and $flag[k] \neq 0$, and there are no processes at line 3

Table B.1: *Experimental Results. The entries marked with "$\star$" have been verified using the optimizations described in Section B.6.1. The other times are copied from [AJRS06], but it is reasonable to expect a similar speed up (say 25%).*

(using $\mathcal{B} = \{\alpha_2(j) \mid j \in \mathbf{N}\}$ where α_2 is the action of line 2) suffices to prove that G'' is terminating. In fact, G'' is the exact set of reachable states for when k is at line 2, which establishes that whenever process k is at line 2 it will reach the critical section.

Results The verification results of our implementation are presented in Table B.1. The models are available in [Nil05]. We have computed sets of states — so-called *starvation free states* — from which starvation freedom for process k under weak fairness is guaranteed, as a set which depends on k. In all cases, the starvation free states contain all of $\Diamond F_k$, to our knowledge. For example, the starvation free states of Szymanski's algorithm are: "whenever process k is at line 2". The column "Time" contains time measured from our implementation. The experiments were run on a PC with a 2.4 GHz processor and 1 GB of RAM.

For the first three protocols, we need apply only Corollary B.5. For the last three protocols, we need also Theorem B.7 — first one application of Corollary B.5 (three for Szymanski), and thereafter one application of Theorem B.7 (four for Dijkstra).

A Case Study We take a closer look at the verification of Szymanski's and Dijkstra's algorithms, to explain what takes time. We present measurements for these algorithms in Tables B.2 and B.3. Their pseudo code is shown in Figures B.8 and B.10.

```
1:    flag[i] := 1
2:    if p ≠ i then
          await flag[p] = 0 then
3:            p := i
4:    flag[i] := 2
5:    if ∃j ≠ i : flag[j] = 2 then goto 1
6:    flag[i] := 0 ; goto 1
```

Figure B.10: Dijkstra's algorithm. Pseudo code for process i.

The verification was done as follows (see also Tables B.2 and B.3). The "preprocessing phase" consists of (1) accelerating the program actions, (2) computing the invariant (the reachability computation shown in the tables), (3) restricting the domain of the actions to $\neg F$ and to the invariant, (4) again accelerating the restricted actions — hence, in total two acceleration phases and one reachability computation. After the preprocessing phase, follow three applications of Corollary B.5 for Szymanski's algorithm, and one for Dijkstra's. Each such application consists of: (1) computing the helpful states (one reachability computation per action), (2) accelerating the corresponding convergence relations, and (3) computing an under-approximation of $\Diamond F$ using backward reachability with the convergence relations. Finally, we apply Theorem B.7 once (with $\mathcal{B} = \{\alpha_2(i) \mid i \in \mathbf{N}\}$) for Szymanski's algorithm, and four times for Dijkstra's algorithm (with $\mathcal{B} = \{\alpha(j) \mid j \in \mathbf{N}, j \neq k\}$ for $\alpha = \alpha_1, \alpha_6, \alpha_5, \alpha_4$).

Alternatively, we can verify Dijkstra's algorithm by replacing the application of Corollary B.5 by applications of Theorem B.7 on $\mathcal{B} = \{\alpha(k)\}$ (in effect "backing" the process k to terminating states), and then proceeding as before. This gives us a total verification time of 38 min 43 s, of which Theorem B.7 takes up 23 min 29 s. The preprocessing phase is then as before.

The "Total" time is the total time for the phase in question. It is slightly higher than the sum of the acceleration and reachability times, as other computations also take place (e.g., computation of left moving states). The "Completion" row shows the total verification time; again, slightly higher than the sum of the presented parts, for the same reason.

No acceleration takes place when we apply Theorem B.7, since we reuse the accelerations made in the preprocessing phase. For Szymanski's algorithm, the 40 s of Theorem B.7 are distributed as follows: computing $\mathit{After}(\mathcal{B}, F)$, 2 s; computing $\mathit{Persist}(\alpha, F)$, 0 s; and computing $\neg\, \mathit{Twice}(\alpha, F)$, 36 s. The computation of $\neg\, \mathit{Twice}(\alpha, F)$ is the most time

Phase · Task	Acceleration	Reachability	Total
Preprocessing	two: 32 s and 44 s	one: 1 s	79 s
3 × Corollary B.5	three: 24 s, 13 s, 28 s	17: 130 s	217 s
1 × Theorem B.7	none	three: 38 s	40 s
Completion			5 min 45 s

Table B.2: *Overview of the verification of Szymanski's algorithm. The Acceleration column shows both the number of accelerations and the time each acceleration took. The same holds for the Reachability column.*

Phase · Task	Acceleration	Reachability	Total
Preprocessing	two: 24 s and 860 s	one: 4 s	888 s
1 × Corollary B.5	one: 357 s	six: 461 s	859 s
4 × Theorem B.7	none	4×3: 330 s, 327 s, 253 s, 260 s	1177 s
Completion			49 min 9 s

Table B.3: *Overview of the verification of Dijkstra's algorithm. The Acceleration column shows both the number of accelerations and the time each acceleration took. The same holds for the Reachability column.*

consuming part by far also for Dijkstra's algorithm (in total 1127 out of 1170 s).

We note that the acceleration of the restricted actions for Dijkstra's algorithm is very expensive (860 s). For example, the first action $\alpha_1(i)$ of Dijkstra's algorithm, which for process i changes the pc from 1 to 2 and sets the flag to 1, takes 2 s to accelerate unrestricted, 76 s when restricted to $\neg F$, 213 s restricted to the invariant, and 166 s restricted to both $\neg F$ and the invariant. Similarly, the action $\alpha_3(i)$ which for process i changes the pc from 3 to 4 and sets the global variable p to i, takes 2 s to accelerate unrestricted, 467 s when restricted to the invariant, 468 s when restricted to both $\neg F$ and the invariant, and the acceleration when restricted to only $\neg F$ was aborted after 30 minutes.

Acceleration of, and reachability computations with, large actions (i.e., whose automata representation is large — see e.g. [AJNS04]) is clearly the bottle-neck.

Discussion Quick acceleration techniques exist for most actions occuring in the considered protocols — referring to actions for which the transitive closure is known, in advance, as a function of the action —

for such actions we can essentially eliminate the time spent on acceleration. It is however motivated to develop quick acceleration techniques for actions with global variables, such as α_3 of Dijkstra's algorithm.

Whenever possible, large actions (meaning, whose automata represenation is large) should be avoided. Actions typically become large when restricted to a "complex" domain or range.

While in Sections B.4 and B.6 we often say that actions should be restricted to $\neg F$, the results are still sound without that restriction — we just risk proving termination for a smaller set of states. If we do not restrict the actions to Inv, we may end up proving termination for unreachable states as well, which is meaningless but sound. In the following we present an optimization which allows us to avoid restricting the actions, without introducing any approximation.

We observe that it is sometimes not necessary to restrict the actions, as the backward reachable states are already in the desired set. This is simple and efficient to check, yet potentially saves much acceleration and reachability computation time. We say that a set of states S is *backward stable (with respect to α)* if:

$$Pre(\alpha, S) \subseteq S\,.$$

Given a set of states S and a set of actions $\mathcal{B}$ for which S is backward stable. The point of backward stableness, is simply that

$$Pre^\star(\mathcal{B}, S) \subseteq S\,.$$

We can use this to avoid potentially costly accelerations, and reachability computations, by not restricting an action to $Inv \cap \neg F$, if that set is anyway backward stable with respect to the action. Better yet, it suffices that $Inv \cap Pre(\alpha, S) \subseteq S$ where Inv is the invariant. The saving essentially comes from not having to compute $((Inv \cap \neg F) \restriction \alpha)^+$, and instead compute just α^+, and thereafter e.g. $Inv \restriction \alpha^+$. However, reachability computations are also faster, as we will demonstrate (later, see Table B.4).

In practice, we expect that almost all actions (typically, all but one) are backward stable with respect to $Inv \cap \neg F$, because not being so means that it is possible to leave F by executing the action. In our experiments, F is the set "process k is in the critical section", and therefore the only action which is not backward stable, should be the parameterized action α such that $\alpha(k)$ leaves F (takes process k out of the critical section).

The optimization to keep the actions small, described above, pays off well. Motivating measurements are shown in Table B.4.

Finally, we present one more optimization, which we however have not implemented. We can speed up the computation of $Twice(\alpha, F)$ at

Model	Szymanski	Dijkstra
Backward stable actions	all except α_7	all except α_6
Preprocessing acceleration	32 s and 16 s (cf. 44 s)	24 s and 27 s (cf. 860 s)
Total verification time	5 min 12 s (cf. 5 min 45 s)	24 min 38 s (cf. 38 min 43 s)

Table B.4: *Verification of Szymanski's and Dijkstra's algorithms, when optimizing the acceleration done in the preprocessing phase. The times are compared with Tables B.2 and B.3. Note that the comparison is made with the second acceleration of the two (the first is as before). Actions for which* $Inv \cap \neg F$ *is backward stable need not be restricted to* $Inv \cap \neg F$ *before acceleration, as described in the discussion. The verification of Dijkstra's algorithm was done using only Theorem B.7, which was suggested as an alternative in the discussion.*

least when $\mathcal{B} = \{\alpha(i) \mid i \in \mathbf{N}\}$ using the following over-approximation: if we know the invariant, we can compute an over-approximation of e.g. $Twice(\alpha(i), F)$ by iterating with the *local transition relation* $\longrightarrow(i) = \{\alpha(i) \mid \alpha \subseteq \longrightarrow\}$ and intersecting with the invariant after each step. In other words,

$$Twice(\alpha(i), F) \supseteq LocalApprox(\alpha(i), F)$$

where the latter set is computed in a finite number of steps as follows:

(1) $(S := S \cup Post(\longrightarrow(i), S) \cap Inv)^+$
(2) $S := Post(\alpha(i), Inv)$
(3) $(S := S \cup Post(\longrightarrow(i), S) \cap Inv)^+$
(4) $S := Post(\alpha(i), Inv)$
(5) $(S := S \cup Post(\longrightarrow(i), S) \cap Inv)^+$

as usual restricting ourselves to outside of F. The meaning of a computation marked with a "+" above is to take the fixed point of S under the operation. These computations must terminate if the set of local states of process i is finite (in at most as many steps as the number of local states). The idea behind the optimization is essentially to replace reachability computations, which involve a particular number of applications of a specific action, with local fixed points. Let us call this optimization *local reachability*. Note the similarity with Proposition B.10. We believe this is useful for our benchmark at least. The speed up should be significant, as we replace the most expensive computations with an elegant computation which does not use reachability at all.

Comparison with Related Work Several works have considered verification of individual starvation freedom for parameterized mutual exclusion protocols. In papers [PXZ02, BHV04] Szymanski's protocol and the Bakery protocol are verified in 95.87 seconds and 9 seconds respectively, using manually supplied abstractions. The works [FPPZ04a, FPPZ04b] verify the Bakery protocol using automatically generated ranking functions, but do not report running times. We have previously verified the Bakery protocol in 44.2 seconds using repeated reachability [Nil05], on the same system.

To our knowledge, starvation freedom for Burns' and Dijkstra's algorithms has not been automatically verified before, except recently in [JS07]. There liveness was proven for the parameterized programs considered in this paper, using transitive closure computation. The times obtained were: Bakery, 13 s; Szymanski, 22 min 49 s; Burns, 98 s; and Dijkstra, 6 min 4 s. Thus, the verification was sometimes faster there. In that work a quicker acceleration scheme was used, but that alone does not explain the difference. We have investigated the reason, and conclude that it is also due to the cost of reachability with relatively large actions — our techniques require many computations of reachable states, rather than one computation of the reachable loops. Techniques exist for quicker accelerations, which can drastically reduce the acceleration time ([ABJN99, PS00, JS07]).

Our techniques cannot find counterexamples, as they are not complete. However, one advantage is that fairness is not explicitly encoded, and therefore the problem setting is much like that of safety analysis, for which many efficient techniques exist [CGJ$^+$03]. For example, over-approximation of reachable states is sound in our setting, in the sense that if we over-approximate the backward reachable states used in our computations, their complements will be under-approximations, but we only risk proving termination for a smaller set of states.

B.7 Conclusions

We have presented techniques for proving liveness and termination properties of fair concurrent programs using backward reachability analysis. The techniques use neither computation of transitive closure nor explicit construction of ranking functions and helpful directions, and relies instead on showing certain commutativity properties between different actions of the program, as well as computing sets of states reachable by certain transition relations (which are subsets of the transition relation of the program).

The advantage of our techniques is that reachability analysis can typically be expected to be simpler to perform than computation of tran-

sitive closures or ranking functions. We further note that the transition relations used for the reachability analysis do not encode fairness explicitly — they are merely subsets of the original transition relation. Because our techniques rely on computing complements of sets of states using backward reachability, over-approximating the set of backward reachable states is sound (but rather than "spuriousness", we risk that the computed set of terminating states becomes too small). A priori, any method successful for the analysis of safety can be used together with our techniques.

We expect that it should be possible to use and develop powerful techniques for backward reachability analysis for many classes of parameterized and infinite-state programs. While our techniques are in general incomplete, their power can be increased by performing repeated applications or by applying complementary techniques afterwards. The examples in the paper indicate that the method should be applicable to several classes of infinite-state systems. In particular, we have shown that our technique is able to prove starvation-freedom for several parameterized mutual exclusion protocols, for which liveness is relatively difficult.

B.8 Bibliography

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

[ACABJ04] Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. Using forward reachability analysis for verification of lossy channel systems. *Formal Methods in System Design*, 25(1):39–65, 2004.

[AČJYK00] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[AJ96a] Parosh Aziz Abdulla and Bengt Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1):71–90, 1996.

[AJ96b] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[AJN+04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. Regular model checking for LTL(MSO). In *Proc.* 16^{th} *Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, 2004.

[AJNS04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A survey of regular model checking. In *Proc. CONCUR 2004,* 14^{th} *Int. Conf. on Concurrency Theory*, volume 3170 of *LNCS*, pages 35–48, 2004.

[AJRS06] Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena. Proving liveness by backwards reachability. In *Proceedings CONCUR 2006, 17th International Conference on Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 95–109. Springer Verlag, 2006.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, 1996.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc.* 16^{th} *Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, 2004.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001*, pages 203–213, 2001.

[BMS05a] A.R. Bradley, Z. Manna, and H.B. Sipma. Linear ranking with reachability. In M. Abadi and L. de Alfaro, editors, *Proc. CONCUR 2005,* 15^{th} *Int. Conf. on Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 491–504, 2005.

[BMS05b] A.R. Bradley, Z. Manna, and H.B. Sipma. Termination analysis of integer linear loops. In K. Etessami and S.K. Rajamani, editors, *Proc.* 17^{th} *Int. Conf. on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 488–502, 2005.

[BMS05c] A.R. Bradley, Z. Manna, and H.B. Sipma. Termination of polynomial programs. In R. Cousot, editor, *Proc. VMCAI 2005, Verification, Model Checking, and Abstract Interpretation,*

6th International Conference, Paris, January 17-19, volume 3385 of *Lecture Notes in Computer Science*, pages 113–129. Springer Verlag, 2005.

[BPZ05] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 164–180. Springer Verlag, 2005.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGMP99] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2:279–287, 1999.

[Cou05] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer Verlag, 2005.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In C. Hankin and I. Siveroni, editors, *Proc.* 12^{th} *Int. Symp. on Static Analysis*, volume 3672 of *LNCS*, pages 87–101. Springer Verlag, 2005.

[CS01] M.A. Colon and H. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Proc. TACAS '01,* 7^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer Verlag, 2001.

[CS02] M.A. Colon and H. Sipma. Practical methods for proving program termination. In Brinskma and Larsen, editors, *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer Verlag, 2002.

[FPPZ04a] Y. Fang, N. Piterman, A. Pnueli, and L.D. Zuck. Liveness with incomprehensible ranking. In K. Jensen and A. Podelski, editors, *Proc. TACAS '04,* 10^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 482–496. Springer Verlag, 2004.

[FPPZ04b] Y. Fang, N. Piterman, A. Pnueli, and L.D. Zuck. Liveness with invisible ranking. In B. Steffen and G. Leiv, editors, *Proc. VMCAI 2004, Verification, Model Checking, and Abstract Interpretation, 5th International Conference, Venice, January 11-13*, volume 2937 of *Lecture Notes in Computer Science*, pages 223–238. Springer Verlag, 2004.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[JS07] Bengt Jonsson and Mayank Saksena. Systematic acceleration in regular model checking. In *Proceedings 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 131–144. Springer Verlag, 2007.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proc.* 28^{th} *ACM Symp. on Principles of Programming Languages*, pages 81–92, 2001.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(4):257–289, 1984.

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proc.* 9^{th} *ACM Symp. on Principles of Distributed Computing, Canada*, pages 377–408, 1990.

[MP91] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R.F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 125–159. ACM Press and Addison-Wesley, 1991.

[Nil05] M. Nilsson. *Regular Model Checking*. PhD thesis, Uppsala University, 2005.

[PPR05] A. Pnueli, Andreas Podelski, and Andrey Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In Nicolas Halbwachs and Lenore Zuck, editors, *Proc. TACAS '05,* 11^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 124–139. Springer Verlag, 2005.

[PR04] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proc. LICS' 04* 20^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 32–41, 2004.

[PR05] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *Proc.* 32^{th} *ACM Symp. on Principles of Programming Languages*, pages 132–144, 2005.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer Verlag, 2000.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[Szy90] B. K. Szymanski. Mutual exclusion revisited. In *Proc. Fifth Jerusalem Conference on Information Technology*, pages 110–117, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[Var91] Moshe Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1–2):79–98, 1991.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

Paper C

Systematic Acceleration in Regular Model Checking

Reprinted from LNCS, volume 4590.

Systematic Acceleration in Regular Model Checking

Bengt Jonsson
bengt@it.uu.se
Uppsala University

Mayank Saksena
mayanks@it.uu.se
Uppsala University

Abstract

Regular model checking is a form of symbolic model checking technique for systems whose states can be represented as finite words over a finite alphabet, where regular sets are used as symbolic representation. A major problem in symbolic model checking of parameterized and infinite-state systems is that fixpoint computations to generate the set of reachable states or the set of reachable loops do not terminate in general. Therefore, *acceleration* techniques have been developed, which calculate the effect of arbitrarily long sequences of transitions generated by some action. We present a systematic method for using acceleration in regular model checking, for the case where each transition changes at most one position in the word; this includes many parameterized algorithms and algorithms on data structures. The method extracts a maximal (in a certain sense) set of actions from a transition relation. These actions, and systematically obtained compositions of them, are accelerated to speed up a fixpoint computation. The extraction can be done on any representation of the transition relation, e.g., as a union of actions or as a single monolithic transducer. Using this approach, we are for the first time able to verify completely automatically both safety and absence of starvation properties for a collection of parameterized synchronization protocols from the literature; for some protocols, we obtain significant improvements in verification time. The results show that symbolic state-space exploration, without using abstractions, is a viable alternative for verification of parameterized systems with a linear topology.

C.1 Introduction

A major approach in algorithmic verification of parameterized and infinite-state systems is to extend the paradigm of symbolic model checking [BCMD92] by appropriate symbolic representations; examples include Petri nets, timed automata, systems with unbounded

communication channels, integers and reals. One direction is *regular model checking*, which considers systems whose states can be represented as finite words over a finite alphabet; regular sets are used to represent sets of states and transition relations. Regular model checking has been proposed as a uniform paradigm for algorithmic verification of several classes of parameterized and infinite-state systems [KMM+01, WB98, BJNT00, AJNS04].

In symbolic model checking of parameterized and infinite-state systems, a major problem is that fixpoint computations that generate the set of reachable states or the set of reachable loops (for verifying liveness properties) do not terminate in general, since there is no uniform bound on the distance (in number of transitions) from an initial configuration to any reachable configuration. To make fixpoint computations converge more frequently, *acceleration* techniques have been developed, which calculate the effect of arbitrarily long sequences of transitions generated by some action (i.e., a subset of the transition relation). This has been done, e.g., for systems with unbounded FIFO channels [BG96, BGWW97, BH99, ACABJ04], systems with counters [BW94, CJ98], and for parameterized systems [ABJN99]. Acceleration is typically applied to small actions, e.g., corresponding to a single program statement or simple loop, since acceleration of larger actions or the entire transition relation is often intractable. Fixpoint computations can be sped up by using accelerated actions in each iteration, thereby allowing the fixpoint computation to converge in many practical cases (e.g., [ACABJ04]).

For regular model checking, methods have been developed for computing the set of reachable configurations or reachable loops [JN00, BJNT00, DLS02, AJNd03]. These algorithms typically work well for small system models, but have difficulties to cope with large transition relations. For instance, the automata-theoretic approach for parameterized systems [AJN+04] transforms verification of a liveness property into the problem of finding reachable loops for a system with a rather large transition relation. There has been no systematic way to to extract actions for acceleration from such a transition relation, and therefore liveness properties for several parameterized mutual exclusion protocols have not been proven automatically by this class of techniques.

In this paper, we present a systematic approach for using acceleration to speed up fixpoint computations in regular model checking. We consider *unary* systems, in which each computation step changes at most one position in the word; many models of parameterized algorithms and algorithms on data structures are unary. Our approach is based on accelerating a class of actions (called *separable*) which can be efficiently accelerated. We present techniques for

(a) systematically extracting a set of separable actions which is maximal in the sense that any other separable action is included in some extracted one; the extraction can be done on any representation of the transition relation, e.g., as a union of actions or as a single monolithic transducer,

(b) systematically composing actions to form separable actions that represent the effect of several transitions; such compositions are analogous to program loops; many verification examples require the acceleration of such compositions, rather than single actions, for termination.

We have implemented our approach in the context of our LTL(MSO) model checker for parameterized systems [AJN+04], and verified safety and liveness properties of several idealized parameterized protocols from the literature, including parameterized algorithms for mutual exclusion (e.g., the Bakery algorithm by Lamport, algorithms by Burns, Szymanski, and Dijkstra). The most important result is that, for the first time, liveness properties have been successfully verified for all of these algorithms; previous approaches have not been successfully applied to all of them. One should also note that our verification, following the automata theoretic approach, does not employ any form of abstraction: it computes an exact representation of the set of reachable states and reachable loops.

Related Work. Works on acceleration techniques in other contexts include techniques for systems with FIFO channels [ACABJ04, BGWW97, BH99] and systems with counter variables [AAB00, WB98]. Finkel, Leroux and colleagues have presented a systematic framework for acceleration techniques for programs with a finite number of variables, typically ranging over integers [BFLS05, FL02]. Their approach cannot be used for regular model checking, in which systems can not be modeled by a fixed number of integer variables. For regular model checking, Pnueli and Shahar [PS00] show how specific acceleration schemes can be defined in a version of S1S. They did not consider composition of actions, which is necessary in many cases, and they have reported verification of liveness for only one example, after applying a manually supplied abstraction. In our earlier work [ABJN99], we proved safety properties of several parameterized protocols by accelerating individual actions; this approach did not consider composition of actions and would therefore not have been able to verify liveness properties.

Proving liveness properties of parameterized systems has been considered also in other approaches. Pnueli, Xu, and Zuck [PXZ02] use a version of counter abstraction to prove absence of starvation properties

for Szymanski's algorithm and the Bakery algorithm. Their abstractions are rather coarse, and lose information so that, e.g., safety properties can no longer be checked. Fang, Piterman, Pnueli, and Zuck [FPPZ04b, FPPZ04a] infer a ranking function and helpful directions of a certain form, by generalizing from the verification of finite instances. These approaches require that a system can be verified using assertions of a certain form. In our earlier work [AJSR06], we proved liveness properties by backwards reachability analysis from "terminated" configurations; this technique can be combined with other techniques for proving liveness, but can not be used to find counterexamples (bugs); our technique is based on state-space exploration, which is guaranteed to report counterexamples when they exist.

Abdulla et al. [ADHR07] verify safety properties of parameterized protocols by over-approximation of backwards reachable states; their approach can not be used for proving liveness properties. Other works apply abstraction [BHV04] or regular inference [HV05] directly on the automata that represent reachable states or the transition relation.

Outline. In the next section, we introduce the framework of regular model checking and the fixpoint computations that are our concern. Section C.3 presents our technique for extracting parts of a transition relation for acceleration. In Section C.4, we present how to use our systematic acceleration in the verification of liveness properties. Experimental results from our implementation, and comparisons with other results are presented in Section C.5. Section C.6 presents conclusions and future work directions.

C.2 The Regular Model Checking Framework

Let Σ be a finite alphabet. A relation $\mathcal{R}$ on Σ^* (the set of finite words over Σ) is *length-preserving* if w and w' are of equal length whenever $(w, w') \in \mathcal{R}$. In this paper, we will only consider length-preserving relations on Σ^*. A relation $\mathcal{R}$ on Σ^* is *regular* if the set $\{(a_1, a_1') \cdots (a_n, a_n') \mid (a_1 \cdots a_n, a_1' \cdots a_n') \in \mathcal{R}\}$ is a regular subset of $(\Sigma \times \Sigma)^*$. A regular relation Σ^* can be represented by a finite-state transducer, i.e., a finite automaton over $(\Sigma \times \Sigma)$.

Regular relations are closed under union $\cup$, intersection $\cap$, relational composition $\circ$, as well as concatenation $\cdot$ defined by $\mathcal{R} \cdot \mathcal{R}' \stackrel{\triangle}{=} \{(w_1 \cdot w_1', w_2 \cdot w_2') \mid w_1 \ \mathcal{R} \ w_2 \text{ and } w_1' \ \mathcal{R}' \ w_2'\}$. For a (regular) set $\mathcal{S}$ of words, let $\mathcal{S} \circ \mathcal{R}$ denote the (regular) set $\{w \mid \exists w' \in \mathcal{S}.\ w' \ \mathcal{R} \ w\}$. We use $\mathcal{R}^+$ to denote the (not necessarily regular) transitive closure of $\mathcal{R}$; and $\mathcal{R}^*$ the reflexive-transitive closure. We denote by $Id = \{(w, w') \mid w = w'\}$ the identity relation on Σ^*.

Definition C.1 A *regular transition system* (RTS for short) over Σ is a pair $(\mathcal{I}, \mathcal{R})$, where

$\mathcal{I}$ is a regular set over Σ, denoting a set of *initial configurations*, and

$\mathcal{R}$ is a regular relation on Σ^*, denoting the *transition relation*.

A *fair regular transition system* (FRTS for short) over Σ is a tuple $(\mathcal{I}, \mathcal{R}, \mathcal{F})$, where $(\mathcal{I}, \mathcal{R})$ is an RTS and $\mathcal{F}$ is a regular set over Σ, denoting the set of *accepting configurations*. Transition relations and regular sets are typically represented by transducers and automata, or by regular expressions. □

A *configuration* w of an RTS $(\mathcal{I}, \mathcal{R})$ is a word $a_1\ a_2\ \cdots\ a_n \in \Sigma^*$. A *computation* of $(\mathcal{I}, \mathcal{R})$ is a finite or infinite sequence $w^0, w^1, w^2, \ldots$ of configurations such that $w^0 \in \mathcal{I}$ and $w^i \mathcal{R} w^{i+1}$ for all adjacent pairs of configurations. A configuration is *reachable* if it occurs in some computation. An infinite computation $w^0, w^1, w^2, \ldots$ of a FRTS is *accepting* if $w^i \in \mathcal{F}$ for infinitely many i.

Many parameterized systems with linear or ring-shaped topologies can be modeled as regular transition systems, by letting each position in a configuration model the local state of a system component. As an example of a parameterized system, we describe the mutual exclusion algorithm by Burns. In the algorithm, an arbitrary number of processes compete for a critical section. The processes are numbered, say from 1 to N. The *local state* of each process consists of a control state ranging over the integers from 1 to 7 and one Boolean flag, $flag$. A pseudo-code description of the behavior of process number i is shown in Figure C.1. For instance, according to the code on line 4, if the control state of a

1:	$flag[i] := 0$
2:	if $\exists j < i : flag[j] = 1$ then goto 1
3:	$flag[i] := 1$
4:	if $\exists j < i : flag[j] = 1$ then goto 1
5:	await $\forall j > i : flag[j] \neq 1$
6:	$flag[i] := 0$
7:	goto 1

Figure C.1: Burns' mutual exclusion algorithm. Pseudo code for process i.

process i is 4, and if the value of $flag$ is 1 for some process $j < i$, then the control state of i may be changed to 1; otherwise to 5. Line 7 represents the critical section.

To model Burns' algorithm as an RTS, we let Σ be the set of possible local states, e.g., represented as tuples $\langle pc, flag \rangle$. A system configuration

is a word in Σ^*. The effect of line j can be represented by a regular relation α_j. For instance, α_1 corresponds to $Id \cdot [(pc = 1) \longrightarrow (pc := 2, flag := 0)] \cdot Id$ where the notation $(pc = 1) \longrightarrow (pc := 2, flag := 0)$ represents the relation $\{(\langle pc_1, flag_1\rangle, \langle pc_2, flag_2\rangle) \mid pc_1 = 1, pc_2 = 2 \text{ and } flag_2 = 0\}$. To distinguish between branches, let α_{ja}, α_{jb} denote the *if* and *else* branch of α_j, for $j = 2, 4$.

It is also possible to model programs that operate on linear unbounded data structures such as queues, stacks, integers, etc. For instance, a stack can be modeled by letting each position in the word represent a position in the stack. The stack should initially contain an arbitrary but bounded number of empty stack positions, which are "statically allocated". We can then faithfully model all finite computations of the system, by initially allocating sufficiently many empty stack positions. We will consider two verification problems:

Reachability: Compute the set of reachable states of a given RTS $(\mathcal{I}, \mathcal{R})$, i.e., the set $\mathcal{I} \circ \mathcal{R}^*$. The problem of verifying any safety property can in the standard way be reduced to that of computing the set of reachable states of a suitable RTS.

Repeated Reachability: Does a given FRTS $(\mathcal{I}, \mathcal{R}, \mathcal{F})$ have an infinite accepting computation? The problem of verifying a liveness properties can, using the classical automata-theoretic framework [VW86] adapted to regular model checking [AJN$^+$04], be reduced to the problem of repeated reachability of a suitable FRTS. A repeated reachability problem can be checked by computing the transitive closure of a transition relation, to be described in Section C.4.

In general, these problems are undecidable, but techniques have been developed which are complete for certain classes of RTSs, and also verify examples from the literature (e.g., [JN00, AJNd03]).

C.3 Verification using Acceleration

We can attempt to compute both reachable and repeatedly reachable configurations by standard fixpoint iterations. Let us describe this for the case of reachability. A naive computation of the set $\mathcal{I} \circ \mathcal{R}^*$ of reachable states is to compute the sequence $\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \cdots$, where $\mathcal{C}_0 = \mathcal{I}$ and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup (\mathcal{C}_i \circ \mathcal{R})$, until a fixpoint is reached, i.e., $\mathcal{C}_{k+1} = \mathcal{C}_k$ for some k. This approach is guaranteed to terminate for finite-state systems, but not in general for parameterized and infinite-state systems, since there is no uniform bound on the number of computation steps needed to reach any particular configuration. For RTSs, $\mathcal{I} \circ \mathcal{R}^*$ and $\mathcal{R}^+$ are in general not computable, but incomplete techniques have been developed [AJNd02,

BJNT00, DLS02], which are guaranteed to complete under conditions which are typically satisfied when $\mathcal{R}$ is "simple", but not when $\mathcal{R}$ is the entire transition relation of an RTS. We therefore present a method to compute $\mathcal{I} \circ \mathcal{R}^*$ or $\mathcal{R}^+$ by decomposing $\mathcal{R}$ into "simple" parts, compute the transitive closure of each part, and then use the results in a refined fixpoint computation.

To this end, let an *action* of the RTS $(\mathcal{I}, \mathcal{R})$ be any subset of $\mathcal{R}$. We use α to range over actions. By *acceleration*, we mean to compute α^+ from α. The fixpoint computation described in the previous paragraph is modified by instead defining $\mathcal{C}_{i+1}$ as the result of choosing an appropriate $\alpha_i \subseteq R^+$, and letting $\mathcal{C}_{i+1} = \mathcal{C}_i \cup (\mathcal{C}_i \circ \alpha_i^+)$. The test for convergence remains the same: is $\mathcal{C}_i = \mathcal{C}_i \cup (\mathcal{C}_i \circ \mathcal{R})$? The main problem is to decide how to choose the sequence of actions $\alpha_0\ \alpha_1\ \cdots$ to accelerate, in order to converge at $\mathcal{I} \circ \mathcal{R}^*$.

We will consider the class of *unary* RTS, in which each computation step changes at most one position in a configuration. This class contains many parameterized synchronization algorithms. For unary RTSs, there is a particular class of actions (called *separable*) which can be accelerated efficiently.

Definition C.2 *A regular relation $\mathcal{R}$ is* unary *if w and w' differ in at most one position whenever $w\ \mathcal{R}\ w'$. A RTS $(\mathcal{I}, \mathcal{R})$ is* unary *if $\mathcal{R}$ is unary. A unary relation is* separable *if it is of form $\phi_L \cdot \tau \cdot \phi_R$, where $\phi_L, \phi_R \subseteq Id$, and τ is a relation on Σ. We call ϕ_L the* left context *and ϕ_R the* right context *of $\phi_L \cdot \tau \cdot \phi_R$.*

Separable unary actions are interesting, because there are efficient techniques for accelerating them, which are complete when ϕ_L and ϕ_R satisfy certain conditions that hold for a majority of separable unary actions encountered in practice [ABJN99, JN00], and yield good under-approximations otherwise. Our verification strategy is therefore to generate a sequence $\alpha_0\ \alpha_1 \cdots$ of separable unary actions to drive the above modified fixpoint computation. To avoid over-approximation, we must obviously require $\alpha_i \subseteq \mathcal{R}^*$ for each i. To make the fixpoint computation as powerful as possible, we will generate as "large" actions as possible. By this, we will mean that any unary separable action in $\mathcal{R}$ is subsumed. We would also like to require the same for any composition of such actions, but this is not possible, since if α and α' are separable unary actions, then in general $\alpha \circ \alpha'$ is not unary and $\alpha \cup \alpha'$ is not separable. We therefore define restricted versions of these operations, *separable composition* $\circ_s$ and *separable union* $\cup_s$, as follows

$$(\phi_L \cdot \tau \cdot \phi_R)\ \circ_s\ (\phi'_L \cdot \tau' \cdot \phi'_R) \quad \stackrel{\triangle}{=} \quad (\phi_L \cap \phi'_L) \cdot \tau \circ \tau' \cdot (\phi_R \cap \phi'_R)$$

$$(\phi_L \cdot \tau \cdot \phi_R)\ \cup_s\ (\phi'_L \cdot \tau' \cdot \phi'_R) \quad \stackrel{\triangle}{=} \quad (\phi_L \cap \phi'_L) \cdot \tau \cup \tau' \cdot (\phi_R \cap \phi'_R)$$

where the changes in α and α' are constrained to occur in the same position. The resulting actions are separable, and can be efficiently accelerated.

Definition C.3 *Let $\mathcal{R}$ be a regular relation. A set of actions $\mathcal{A}$ is* separable-complete with respect to $\mathcal{R}$*, if it satisfies:*
(U) For any sequence $\alpha_1, \ldots, \alpha_n$ of separable unary actions, where $\alpha_j \subseteq \mathcal{R}$ for $j \in [1, n]$, there is an action $\alpha \in \mathcal{A}$ such that

$$(\alpha_1 \circ_s \ldots \circ_s \alpha_n)^+ \subseteq \alpha^+$$

If condition (U) is true for $n \leq k$, for some bound k, the set is separable-complete up to k*, and k is called the* composition depth. □

As a special case, if $\mathcal{A}$ is separable-complete up to 1, then any separable unary action $\alpha' \subseteq \mathcal{R}$ is subsumed by some $\alpha \in \mathcal{A}$.

Let us see why separable-completeness is relevant for Burns' algorithm. Imagine that we are computing $\mathcal{I} \circ \mathcal{R}^*$ for Burns' algorithm, using a fixpoint computation. Consider a configuration where there are arbitrarily many processes on line 2, each with α_{2b} enabled. It is then possible for any single process to proceed to line 5, via lines 3 and 4. However, whenever α_3 is executed by some process i, all processes $j < i$ are blocked. Hence, in order for arbitrarily many processes to move from 2 to 5, they must act sequentially from higher to lower index. It follows that we need the accelerated sequential composition $(\alpha_{2b} \circ_s \alpha_3 \circ_s \alpha_{4b})^+$, to capture this behaviour; a fixpoint computation using only $\alpha_{2b}^+, \alpha_3^+, \alpha_{4b}^+$ would need unboundedly many computation steps. If (U) were true, we would have an action with $\alpha^+ \supseteq (\alpha_{2b} \circ_s \alpha_3 \circ_s \alpha_{4b})^+$, allowing us to compute the set of reachable configurations.

We are now ready to present our technique for generating actions to be accelerated in the fixpoint computation; it will automatically generate a finite set of actions which is separable-complete.

Generation Procedure. Our procedure for generating a sequence of actions that satisfy condition (U) has three steps.

1. We obtain any finite set of separable actions $\mathcal{A}'$ such that $\mathcal{R} = \cup\mathcal{A}'$.

   One way to do this is to extract such actions from a representation of $\mathcal{R}$ as a minimal deterministic automaton $\mathcal{T} = \langle Q, \Sigma \times \Sigma, s_0, \delta, F \rangle$, as follows. Let $\mathcal{T}(s, Q)$ equal $\mathcal{T}$ but with $s_0 = s$ and $F = Q$. Then $\mathcal{R}$ is the union of actions $\{\phi_L \cdot \tau \cdot \phi_R\}$ where $\phi_L = \mathcal{T}(s_0, \{s\}) \cap Id$, and $\phi_R = \mathcal{T}(t, F) \cap Id$, and $\tau = \delta(s, t)$ for states $s, t \in Q$ (and $\phi_L, \phi_R, \tau \neq \emptyset$).

2. We thereafter transform $\mathcal{A}'$ so that it has the property that any separable unary action $\alpha \subseteq \mathcal{R}$ is *in* (i.e., a subset of) the separable union of some actions in $\mathcal{A}'$. For this purpose, we define two operations on separable unary actions:

$$\begin{array}{ccccc} (\phi_L \cdot \tau \cdot \phi_R) & & \sqcap_L & & (\phi'_L \cdot \tau' \cdot \phi'_R) \\ (\phi_L \cap \phi'_L) & \cdot & (\tau \cap \tau') & \cdot & (\phi_R \cup \phi'_R) \end{array} \triangleq$$

$$\begin{array}{ccccc} (\phi_L \cdot \tau \cdot \phi_R) & & \sqcap_R & & (\phi'_L \cdot \tau' \cdot \phi'_R) \\ (\phi_L \cup \phi'_L) & \cdot & (\tau \cap \tau') & \cdot & (\phi_R \cap \phi'_R) \end{array} \triangleq$$

Closing the set of actions under the operations $\sqcap_L$ and $\sqcap_R$ achieves the goal. As an optimization, we delete actions that are then subsets of other actions.

3. Finally, we close the set of actions $\mathcal{A}'$, from previous step, under $\cup_s$. Again, as an optimization, we delete actions that become subsets of other actions.

We motivate step **2** for Burns' algorithm. Suppose step **1** is applied to a deterministic representation of $\mathcal{R}$. We get $\mathcal{A}' \supseteq \{\alpha, \alpha'\}$, with $\alpha = \phi_{L\,4a} \cdot (\tau_3 \cup \tau) \cdot Id$, and $\alpha' = \phi_{L\,4b} \cdot (\tau_3 \cup \tau') \cdot Id$, for some τ, τ'. The desired property is false: α_3 is not in the separable union of α, α' (nor of $\mathcal{A}'$). The left context of α_3 has been divided. However, $\alpha_3 = (\alpha \sqcap_R \alpha')$, giving the desired property. Without step **2**, our procedure under-approximates α_3 and sequential compositions involving α_3.

The generated actions are separable-complete up to 1 by construction (by steps **2** and **3**). Let us now establish that they are even separable-complete. We use the following lemma, which establishes how $\circ_s$ and $\cup_s$ are related.

Lemma C.4 *Let $\mathcal{A}' = \{\alpha_1, \ldots, \alpha_m\}$ be a set of separable unary actions, with $\alpha_j = \phi_L^j \cdot \tau_j \cdot \phi_R^j$, for $j \in [1, m]$. Let $\sigma = \alpha_{i_1} \circ_s \alpha_{i_2} \circ_s \ldots \circ_s \alpha_{i_n}$ be any composition such that each $\alpha \in \mathcal{A}'$ occurs at least once. Then:*

$$\sigma \subseteq \phi_L^1 \cap \ldots \cap \phi_L^m \cdot (\tau_1 \cup \ldots \cup \tau_m)^+ \cdot \phi_R^1 \cap \ldots \cap \phi_R^m$$

□

Theorem C.5 *The set of actions generated by steps* **1**–**3** *is separable-complete.*

Proof. Given any sequence $\alpha_1, \ldots, \alpha_n$, where $\alpha_j \subseteq \mathcal{R}$ for $j \in [1, n]$. Let us denote the fact that the generated actions have composition depth

1 by (U_1). By (U_1), there are actions $\alpha'_1, \ldots, \alpha'_n$ generated by our procedure such that $\alpha_j \subseteq \alpha'_j$, for each j. Again by (U_1), there exists a generated $\alpha = \phi_L \cdot \tau \cdot \phi_R$ such that $\alpha \supseteq \alpha'_1 \cup_s \ldots \cup_s \alpha'_n$. Now, by the lemma, $\alpha'_1 \circ_s \ldots \circ_s \alpha'_n \subseteq \phi_L \cdot \tau^+ \cdot \phi_R$. Finally, $(\phi_L \cdot \tau^+ \cdot \phi_R)^+ \subseteq \alpha^+$. $\square$

Note on Complexity. Our procedure is essentially conjoining the guards of the actions; so an upper bound of the number of obtained actions is $2^{|\mathcal{A}'|}$, where $\mathcal{A}'$ is the least set satisfying the property of step **2**. For our benchmark (see Section C.5), the actions can only be composed in a monotonic order, so the bound is only $|\mathcal{A}'|^2$. Nonetheless, in practice, we may choose to combine actions under $\cup_s$ a fixed number of iterations in step **3**, obtaining $\mathcal{A}$ with composition depth k.

C.4 Verifying Liveness

In this section, we describe how to verify liveness properties, which are reduced to the repeated reachability problem of a suitable FRTS. In particular, we describe how liveness properties of parameterized algorithms are verified using our $LTL(MSO)$ model checker [AJN+04].

Recall that the falsification of a liveness property can be reduced to checking whether an FRTS has an infinite accepting run. Since the transition relation is length-preserving, so that each computation can visit only a finite set of configurations, this problem can be solved by repeated reachability, i.e., by checking whether there exists a reachable loop containing some configuration from $\mathcal{F}$. This is equivalent to checking whether there is a reachable configuration w in $\mathcal{F}$ such that $(w, w) \in Id \cap \mathcal{R}^+$, which can be checked as follows [Nil05].

(1) Compute the set of *reachable configurations* $Inv = \mathcal{I} \circ \mathcal{R}^*$, as described in Section C.3.

(2) Let $Inv_{\mathcal{F}} = \{(w, w') \mid w \in Inv \cap \mathcal{F}, (w, w') \in \mathcal{R}\}$, i.e., the *relation* containing all pairs of consecutive reachable configurations, where the first satisfies $\mathcal{F}$.

(3) Compute the relation $Inv_{\mathcal{F}} \circ \mathcal{R}^*$ as a fixpoint, which in the acceleration-based version constructs the sequence $\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \cdots$, where $\mathcal{C}_0 = Inv_{\mathcal{F}}$ and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \mathcal{C}_i \circ \alpha_i^+$ for a suitable action $\alpha_i \subseteq \mathcal{R}^+$, until $\mathcal{C}_i = \mathcal{C}_i \cup \mathcal{C}_i \circ \mathcal{R}$.

(4) If the fixpoint computation in (3) converges, a repeatedly reachable configuration w exists if and only if $(Inv_{\mathcal{F}} \circ \mathcal{R}^*) \cap Id$ is non-empty.

Note that if $\mathcal{C}_i \cap Id$ is non-empty for some approximation $\mathcal{C}_i$, we can abort the fixpoint computation of (3), and report that $\mathcal{C}_i$ contains a repeatedly reachable configuration.

The reachability phase (1) computes a fixpoint on sets of configurations, while the repeated reachability phase (3) computes a fixpoint on relations of configurations; the latter is significantly more difficult to compute.

We next show how this procedure specializes to verifying absence of starvation for parameterized systems. A typical liveness property, absence of starvation, is of form $\Box \forall i\, (\phi(i) \longrightarrow \Diamond \psi(i))$, where i ranges over processes modeled by positions in the configuration. For instance, for Burns' algorithm we check the property $\Box \forall i\, ((\mathrm{pc}[i] = 1 \wedge i = 0) \longrightarrow \Diamond \mathrm{pc}[i] = 7)$. This property is proven assuming *weak process fairness*, i.e., that in an infinite computation, each process is infinitely often either blocked or progressing, which can be expressed as $\forall i\, \Box\Diamond(\alpha(i) \vee \neg En(\alpha(i)))$, where $\alpha(i)$ is a disjunction of all actions process i can take, and $En(\alpha(i))$ is true if and only if process i is not blocked.

To verify absence of starvation using the automata-theoretic approach [VW86, AJN$^+$04], the transition relation, fairness requirements and the negation of the liveness properties are conjoined and compiled into an FRTS, which accepts all fair computations of the system which violate the liveness property, i.e., satisfy $\Diamond \exists i\, (\phi(i) \wedge \Box \neg \psi(i))$. The negation of the liveness property is transformed into an extra boolean component $b_{violate}(i)$ in the local state of each position i, such that if $b_{violate}(i)$ is true, then process i satisfies $\Box \neg \psi(i)$. Process i may non-deterministically set $b_{violate}(i)$ to true. The weak fairness requirement is transformed into an extra boolean component $b_{fair}(i)$ in the local state of each position i and the set $\mathcal{F}$ of accepting configurations in which $b_{fair}(i)$ is 1 for all i and $b_{violate}(i)$ is 1 for some i. All components $b_{fair}(i)$ are set to 0 immediately after some configuration in $\mathcal{F}$ was visited, and each $b_{fair}(i)$ is thereafter set to 1 whenever process i satisfies $\alpha(i) \vee \neg En(\alpha(i))$. The repeated reachability problem becomes to check whether there is an infinite computation which first visits a configuration where $b_{violate}(i)$ is 1 for some i, and thereafter infinitely often visits a configuration in $\mathcal{F}$.

The above procedure can be adapted to this setting by inserting a step ($1'$) between steps (1) and (2), which computes the set $Inv' = [Inv \wedge \exists i(\phi(i) \wedge b_{violate}(i))] \circ \mathcal{R}'^{*}$, where $\mathcal{R}'$ is $\mathcal{R}$ constrained to follow the semantics of $b_{violate}$, as described above. For the remaining steps, Inv' and $\mathcal{R}'$ take the roles of Inv and $\mathcal{R}$. For step (3), we further constrain $\mathcal{R}'$ to follow the semantics of b_{fair}. We have also added the following optimizations to our model checker.

- *Separating updates of* $b_{fair}(i)$. We separate the updates of $b_{fair}(i)$ into one action that sets it when $\alpha(i)$ is taken, and one action that sets it when $\neg En(\alpha(i))$; this equivalent modeling makes the acceleration work more efficiently, since actions remain unary.

- *One violating witness.* We constrain the transition relation so that $b_{violate}(i)$ can be true for *at most* one process i; this simplifies the transition relation. Note that this does not forbid other processes from violating the property.

C.5 Experimental Results

We have implemented the systematic acceleration method described in this paper in our $LTL(MSO)$ model checker [AJN+04], and used it to generate the set of reachable states, as described in Section C.3, and to check absence of individual starvation under weak fairness for parameterized synchronization algorithms from the literature, as described in Section C.4. The models are described in detail in [Nil05], and are also available in the Appendix. For the Bakery algorithm, we verified the property $\mathsf{Ba} \triangleq \Box\forall i\,(\mathrm{q}[i] = w \longrightarrow \Diamond \mathrm{q}[i] = cs)$. All other checked liveness properties were of form $\Box\forall i\,(\phi(i) \longrightarrow \Diamond\psi(i))$, where $\psi(i)$, defined as $\mathrm{pc}[i] = cs$, represents that process i is in the critical section, and where $\phi(i)$ expresses that process i intends to reach the critical section, and that also it is reasonable to suspect that process i is guaranteed to succeed in doing so. For each choice of $\phi(i)$ our implementation either reports a success in verification, or a counterexample. We checked several properties, whose $\phi(i)$ are given below, named after the initial letters of their corresponding protocols; Bakery, Burns, Szymanski, Dijkstra.

$$
\begin{array}{lll}
\mathsf{Bu}_1 : \mathrm{pc}[i] = 1 \wedge i = 0 & & \mathsf{Sz}_1 : \mathrm{pc}[i] = 1 \\
\mathsf{Bu}_2 : \mathrm{pc}[i] = 1 \wedge i \neq 0 & & \mathsf{Sz}_2 : \mathrm{pc}[i] = 2 \\
\mathsf{Bu}_3 : \mathrm{pc}[i] \neq 1 \wedge i \neq 0 & & \mathsf{Sz}_3 : \mathrm{pc}[i] \neq 1 \\
\mathsf{Bu}_4 : i = 0 & & \\
\mathsf{Di}_1 : \mathrm{p}[i] \wedge \mathrm{flag}[i] \neq 0 \wedge \forall j \neq i \,.\, \mathrm{pc}[j] \neq 3 & & \\
\mathsf{Di}_2 : \mathrm{p}[i] \wedge \mathrm{flag}[i] = 0 \wedge \forall j \neq i \,.\, \mathrm{pc}[j] \neq 3 & &
\end{array}
$$

We used composition depth k as a parameter, successively using higher values if the verification did not succeed within a certain time bound. The times are given for the best values of k, not including "too low k" time. All protocols worked with some $k \in [2, 5]$. Dijkstra's protocol needed 5, and Szymanski's protocol was significantly slower with $k > 2$, due to its actions using many different guards. If the generated actions become separable-complete for parameter k, using a higher value is not significantly slower, as testing for separable-completeness is quick. By Lemma 1, we need not consider k higher than the number of actions generated in step **2** of the generation procedure – that k gives the best approximation of $\mathcal{R}^+$, but can be suboptimal with respect to time. To handle the fact that one action of Dijkstra's protocol is not unary, we

extended the composition techniques to a class of non-unary actions in the most straight-forward way. The experiments were run on a PC with a 2.4 GHz processor and 1 GB of RAM.

Results and Comparison with Related Work. Our verification results are presented in Tables C.1 and C.2. The table contains time measured in seconds for the analysis, but does not include the translation from LTL(MSO) formulas into FRTSs. False properties, for which a counterexample was found, are marked "(f)". In the table, we compare our times with the works [Nil05, AJN+04, AJSR06], as they use similar techniques, and were in fact timed on the same system. We also present related work, in alphabetical order with respect to authors. Note that works [Nil05, AJN+04, PS00] could only have succeeded to verify Burns' and Dijkstra's protocols if the right sequential compositions were included; but they are difficult to find manually.

- Abdulla et al. [ADHR07] use over-approximation for safety properties, obtaining times an order of magnitude better than ours (0.004–3.9 seconds), but the technique can not be extended to liveness properties.

- The techniques of [AJSR06] compute states which are guaranteed to satisfy $\psi(i)$ using backwards reachability, thus avoiding the repeated reachability problem. However, they are not able to produce counterexamples, and are sometimes slower (due to requiring many accelerations).

- Bouajjani et al. [BHV04] verify liveness of Bakery, as well as safety of all listed protocols, using counter-example guided abstractions, in 0.06–0.73 seconds.

- Fang et al. [FPPZ04a, FPPZ04b, FMPZ06] verify the Bakery protocol using automatically generated ranking functions, but do not report running times.

- The works of Nilsson et al. [Nil05, AJN+04] report times for essentially the same technique, so we gave the best time for each protocol. The verification setting is as ours, but without the systematic addition of sequential compositions.

- Pnueli and Shahar [PS00], use user defined accelerations to verify safety properties of Szymanski's protocol in 0.2 seconds, as well as some protocols not in our benchmark.

- Pnueli et al. [PXZ02] verify liveness of the Bakery and Szymanski protocols using manually supplied counter abstractions, in 1 and

Table C.1: *Liveness. Comparison of verification times with [Nil05, AJN+04, AJSR06].*

Property	This work	[Nil05, AJN+04]	[AJSR06]
Ba	13	23	36
Bu_1	98		450
Bu_2	56 (f)		
Bu_3	60 (f)		
Bu_4	144		
Sz_1	540 (f)		
Sz_2	1369		435
Sz_3	1635		
Di_1	244		3311
Di_2	1031 (f)		

Table C.2: *Safety. Comparison of verification times with [Nil05, AJN+04, AJSR06].*

Protocol	This work	[Nil05, AJN+04]
Bakery	4	5
Burns	15	39
Szymanski	19	34
Dijkstra	25	38

96 seconds respectively. Their modeling of Szymanski's protocol is slightly different from ours, so we can not say which of the true properties were checked.

Using the techniques of this paper, we can compute an exact representation of the reachable loops for all the above protocols. It has, to our knowledge, never been done for Burns' and Dijkstra's protocols before.

C.6 Conclusions and Future Work

We have presented a systematic method for using acceleration to speed up fixpoint computations in regular model checking. The method is defined for unary transition relations, and is independent of how the transition relation is represented. We show how to accelerate a set of actions which is maximal in a certain sense, in order to make the verification

as powerful as possible. Using this approach, we have succeeded in verifying safety and liveness of parameterized synchronization protocols, whose verification has not been reported before.

Our work shows that acceleration-based symbolic state-space exploration can be used efficiently also in regular model checking, thus extending this approach from other classes of systems (e.g., [ACABJ04, BGWW97, BH99, WB98, BFLS05, FL02]). Future work includes extending the approach to non-unary transition relations, in order to handle, e.g., systems with synchronous communication between adjacent processes.

C.7 Bibliography

[AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In E.A. Emerson and P. Sistla, editors, *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 419–434, 2000.

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

[ACABJ04] P.A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using forward reachability analysis for verification of lossy channel systems. *Formal Methods in System Design*, 25(1):39–65, 2004.

[ADHR07] P.A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Regular model checking without transducers. In *Proc. TACAS '07,* 13^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2007. to appear.

[AJN+04] P.A. Abdulla, B. Jonsson, Marcus Nilsson, Julien d'Orso, and M. Saksena. Regular model checking for MSO + LTL. In R. Alur and D. Peled, editors, *Proc.* 16^{th} *Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 348–360, 2004.

[AJNd02] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002,* 13^{th} *Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.

[AJNd03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Algorithmic improvements in regular model checking. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 236–248, 2003.

[AJNS04] P.A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In P. Gardner and N. Yoshida, editors, *Proc. CONCUR 2004,* 15^{th} *Int. Conf. on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer Verlag, 2004.

[AJSR06] P.A. Abdulla, B. Jonsson, M. Saksena, and A. Rezine. Proving liveness by backwards reachability. In C. Baier and H. Hermanns, editors, *Proc. CONCUR 2006,* 17^{th} *Int. Conf. on Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 95–109. Springer Verlag, 2006.

[BCMD92] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BFLS05] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In D. Peled and Y.-K. Tsay, editors, *Proc. ATVA 2005,* 3^{th} *Int. Symp. Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, 1996.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. of the Fourth International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer Verlag, 1997.

[BH99] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1-2):211–250, 1999.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In R. Alur and D. Peled, editors, *Proc.* 16^{th} *Int. Conf. on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386, 2004.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer Verlag, 2000.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc.* 6^{th} *Int. Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer Verlag, 1994.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV'98*. LNCS 1427, 1998.

[DLS02] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. *J. Log. Algebr. Program.*, 52-53:109–127, 2002.

[FL02] S. Finkel and J. Leroux. How to compose presburger-accelerations: Applications to broadcast protocols. In M. Agrawal and A. Seth, editors, *Proc. FSTTCS 2002,* 22^{th} *Conf. Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[FMPZ06] Yi Fang, Kenneth L. McMillan, Amir Pnueli, and Lenore D. Zuck. Liveness by invisible invariants. In *FORTE*, pages 356–371, 2006.

[FPPZ04a] Y. Fang, N. Piterman, A. Pnueli, and L.D. Zuck. Liveness with incomprehensible ranking. In K. Jensen and A. Podelski, editors, *Proc. TACAS '04,* 10^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 482–496. Springer Verlag, 2004.

[FPPZ04b] Y. Fang, N. Piterman, A. Pnueli, and L.D. Zuck. Liveness with invisible ranking. In B. Steffen and G. Leiv, editors, *Proc. VMCAI 2004, Verification, Model Checking, and Abstract Interpretation, 5th International Conference, Venice, January 11-13*, volume 2937 of *Lecture Notes in Computer Science*, pages 223–238. Springer Verlag, 2004.

[HV05] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.*, 138(3):21–36, 2005.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and

M. Schwartzbach, editors, *Proc. TACAS '00,* 6^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, 2000.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.

[Nil05] M. Nilsson. *Regular Model Checking.* PhD thesis, Uppsala University, 2005.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer Verlag, 2000.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *Proc.* 14^{th} *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

C.8 Appendix

We give the pseudo code, and $LTL(MSO)$ models of the mutual exclusion protocols we have verified. A more thorough description of the models are found in the thesis [Nil05].

C.8.1 Protocols

Pseudo code for the Bakery, Szymanski, Burns, and Dijkstra mutual exclusion protocols are given in figures C.2, C.3, C.4, C.5 respectively.

Idle:	$ticket_i := 1 + \mathbf{max}_j\ ticket_j$
Waiting:	**await** $\forall j \neq i : (ticket_i < ticket_j \vee ticket_j = 0)$
Critical:	$ticket_i := 0$

Figure C.2: The bakery algorithm. Pseudo code for process i.

1:	**await** $\forall j : j \neq i : \neg s[j]$
2:	$w[i], s[i] := true, true$
3:	**if** $\exists j : j \neq i : (pc[j] \neq 1) \wedge \neg w[j]$
	then $s[i] := false$; goto 4
	else $w[i] := false$; goto 5
4:	**await** $\exists j : j \neq i : s[j] \wedge \neg w[j]$
	then $w[i], s[i] := false, true$
5:	**await** $\forall j : j \neq i : \neg w[j]$
6:	**await** $\forall j : j < i : \neg s[j]$
7:	$s[i] := false$; goto 1

Figure C.3: Szymanski's algorithm. Pseudo code for process i.

1:	$flag[i] := 0$
2:	**if** $\exists j < i : flag[j] = 1$ **then** goto 1
3:	$flag[i] := 1$
4:	**if** $\exists j < i : flag[j] = 1$ **then** goto 1
5:	**await** $\forall j > i : flag[j] \neq 1$
6:	$flag[i] := 0$
7:	goto 1

Figure C.4: Burns' algorithm. Pseudo code for process i.

```
1:    flag[i] := 1
2:    if p ≠ i then
          await flag[p] = 0 then
3:            p := i
4:    flag[i] := 2
5:    if ∃j ≠ i : flag[j] = 2 then goto 1
6:    flag[i] := 0
7:    goto 1
```

Figure C.5: Dijkstra's algorithm. Pseudo code for process i.

C.8.2 $LTL(MSO)$ models

Models for the Bakery, Szymanski, Burns, and Dijkstra mutual exclusion protocols are given in figures C.6, C.7, C.8, C.9 respectively. We have used "syntactic sugar" to make the models more readable. The expression $En(\alpha)$ represents the guard of α (obtained as the "unprimed" part of α). The following formula definitions are common for all models.

- *idle* $\triangleq$ $\forall i\ copy(i)$
- *copy-other*(i) $\triangleq$ $\forall j \neq i\ copy(j)$
- *sys* $\triangleq$ $initial \wedge \Box(\exists i\ \alpha(i) \vee idle)$
- *safety-check* $\triangleq$ $sys \wedge \neg mutex$
- *fairness* $\triangleq$ $\forall i\ \Box\Diamond(\alpha(i) \vee \neg En(\alpha(i)))$
- *liveness-check* $\triangleq$ $sys \wedge fairness \wedge \neg liveness$

q **array of** $\{O, W, C\}$

$$
\begin{array}{lcl}
copy(i) & \triangleq & q[i] = q'[i] \\
\alpha_1(i) & \triangleq & q[i] = O \wedge q'[i] = W \wedge \forall j > i\ q[j] = O \wedge \\
 & & (i = 0 \vee q[i-1] = W \vee q[i-1] = C) \\
\alpha_2(i) & \triangleq & q[i] = W \wedge q'[i] = C \wedge \forall j < i\ q[j] = O \\
\alpha_3(i) & \triangleq & q[i] = C \wedge q'[i] = O \\
\alpha(i) & \triangleq & copy\text{-}other(i) \wedge (\alpha_1(i) \vee \alpha_2(i) \vee \alpha_3(i)) \\
initial & \triangleq & \forall i\ q[i] = O \\
mutex & \triangleq & \Box\neg\exists i\,\exists j \neq i\ (q[i] = C \wedge q[j] = C) \\
liveness & \triangleq & \Box\,\forall i\ (q[i] = W \longrightarrow \Diamond\ q[i] = C)
\end{array}
$$

Figure C.6: The bakery algorithm. Pseudo code for process i in $LTL(MSO)$.

pc **array of** $\{1, \ldots, 7\}$

$$
\begin{array}{lcl}
\mathit{copy\text{-}w}(i) & \triangleq & w[i] \longleftrightarrow w'[i] \\
\mathit{copy\text{-}s}(i) & \triangleq & s[i] \longleftrightarrow s'[i] \\
\mathit{copy}(i) & \triangleq & pc[i] = pc'[i] \wedge \mathit{copy\text{-}w}(i) \wedge \mathit{copy\text{-}s}(i) \\
\alpha_1(i) & \triangleq & pc[i] = 1 \wedge \forall j \neq i \; \neg s[j] \wedge pc'[i] = 2 \wedge \\
 & & \mathit{copy\text{-}w}(i) \wedge \mathit{copy\text{-}s}(i) \\
\alpha_2(i) & \triangleq & pc[i] = 2 \wedge pc'[i] = 3 \wedge w'[i] \wedge s'[i] \\
\alpha_3(i) & \triangleq & (pc[i] = 3 \wedge \quad \exists j \neq i \; (pc[j] \neq 1 \wedge \neg w[j]) \wedge \\
 & & pc'[i] = 4 \wedge \neg s'[i] \wedge \mathit{copy\text{-}w}(i)) \vee \\
 & & (pc[i] = 3 \wedge \neg \exists j \neq i \; (pc[j] \neq 1 \wedge \neg w[j]) \wedge \\
 & & pc'[i] = 5 \wedge \neg w'[i] \wedge \mathit{copy\text{-}s}(i)) \\
\alpha_4(i) & \triangleq & pc[i] = 4 \wedge \exists j \neq i \; (s[j] \wedge \neg w[j]) \wedge pc'[i] = \\
 & & 5 \wedge \neg w'[i] \wedge s'[i] \\
\alpha_5(i) & \triangleq & pc[i] = 5 \wedge \forall j \neq i \; \neg w[j] \wedge pc'[i] = 6 \wedge \\
 & & \mathit{copy\text{-}w}(i) \wedge \mathit{copy\text{-}s}(i) \\
\alpha_6(i) & \triangleq & pc[i] = 6 \wedge \forall j < i \; \neg s[j] \wedge pc'[i] = 7 \wedge \\
 & & \mathit{copy\text{-}w}(i) \wedge \mathit{copy\text{-}s}(i) \\
\alpha_7(i) & \triangleq & pc[i] = 7 \wedge pc'[i] = 1 \wedge \neg s'[i] \wedge \mathit{copy\text{-}w}(i) \\
\alpha(i) & \triangleq & \mathit{copy\text{-}other}(i) \wedge (\alpha_1(i) \vee \ldots \vee \alpha_7(i)) \\
\mathit{initial} & \triangleq & \forall i \; pc[i] = 1 \wedge \neg s[i] \wedge \neg w[i] \\
\mathit{mutex} & \triangleq & \Box \neg \exists i \, \exists j \neq i \; (pc[i] = 7 \wedge pc[j] = 7) \\
\mathit{liveness}_1 & \triangleq & \Box \, \forall i \; (pc[i] = 1 \longrightarrow \Diamond \; pc[i] = 7) \; \boxed{\text{false}} \\
\mathit{liveness}_2 & \triangleq & \Box \, \forall i \; (pc[i] = 2 \longrightarrow \Diamond \; pc[i] = 7) \; \boxed{\text{true}} \\
\mathit{liveness}_3 & \triangleq & \Box \, \forall i \; (pc[i] \neq 1 \longrightarrow \Diamond \; pc[i] = 7) \; \boxed{\text{true}}
\end{array}
$$

Figure C.7: Szymanski's algorithm. Pseudo code for process i in $LTL(MSO)$.

pc **array of** $\{1, \ldots, 6\}$

copy-flag(i)	$\triangleq$	$flag[i] \longleftrightarrow flag'[i]$
copy(i)	$\triangleq$	$pc[i] = pc'[i] \wedge$ *copy-flag*(i)
$\alpha_1(i)$	$\triangleq$	$pc[i] = 1 \wedge pc'[i] = 2 \wedge \neg flag'[i]$
$\alpha_2(i)$	$\triangleq$	$(pc[i] = 2 \wedge \exists j < i\ flag[j] \wedge pc'[i] = 1 \wedge$ *copy-flag*$(i)) \vee$
		$(pc[i] = 2 \wedge \neg\exists j < i\ flag[j] \wedge pc'[i] = 3 \wedge$ *copy-flag*$(i))$
$\alpha_3(i)$	$\triangleq$	$pc[i] = 3 \wedge pc'[i] = 4 \wedge flag'[i]$
$\alpha_4(i)$	$\triangleq$	$(pc[i] = 4 \wedge \exists j < i\ flag[j] \wedge pc'[i] = 1 \wedge$ *copy-flag*$(i)) \vee$
		$(pc[i] = 4 \wedge \neg\exists j < i\ flag[j] \wedge pc'[i] = 5 \wedge$ *copy-flag*$(i))$
$\alpha_5(i)$	$\triangleq$	$pc[i] = 5 \wedge \forall j > i\ \neg flag[j] \wedge pc'[i] = 6 \wedge$ *copy-flag*(i)
$\alpha_6(i)$	$\triangleq$	$pc[i] = 6 \wedge pc'[i] = 1 \wedge \neg flag'[i]$
$\alpha(i)$	$\triangleq$	*copy-other*$(i) \wedge (\alpha_1(i) \vee \ldots \vee \alpha_6(i))$
initial	$\triangleq$	$\forall i\ pc[i] = 1 \wedge \neg flag[i]$
mutex	$\triangleq$	$\Box\neg\exists i\, \exists j \neq i\ (pc[i] = 6 \wedge pc[j] = 6)$
$liveness_1$	$\triangleq$	$\Box\, \forall i\ ((pc[i] = 1 \wedge i = 0) \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{true}}$
$liveness_2$	$\triangleq$	$\Box\, \forall i\ ((pc[i] = 1 \wedge i \neq 0) \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{false}}$
$liveness_3$	$\triangleq$	$\Box\, \forall i\ ((pc[i] \neq 1 \wedge i \neq 0) \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{false}}$
$liveness_4$	$\triangleq$	$\Box\, \forall i\ (i = 0 \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{true}}$

Figure C.8: Burns' algorithm. Pseudo code for process i in $LTL(MSO)$.

pc **array of** $\{1,\ldots,6\}$, $flag$ **array of** $\{0,1,2\}$

p-set(i)	$\triangleq$	$\forall j\ p'[j] \longleftrightarrow j = i$
flag-zero-at-p	$\triangleq$	$\forall i\ p[i] \longrightarrow flag[i] = 0$
copy(i)	$\triangleq$	$pc[i] = pc'[i] \wedge flag[i] = flag'[i] \wedge (p[i] \longleftrightarrow p'[i])$
copy-other-except-p(i)	$\triangleq$	$\forall j \neq i\ pc[j] = pc'[j] \wedge flag[j] = flag'[j]$
copy-p	$\triangleq$	$\forall i\ p[i] \longleftrightarrow p'[i]$
copy-flag(i)	$\triangleq$	$flag[i] = flag'[i]$
$\alpha_1(i)$	$\triangleq$	$pc[i] = 1 \wedge flag'[i] = 1 \wedge pc'[i] = 2 \wedge$ *copy-p*
$\alpha_2(i)$	$\triangleq$	$(pc[i] = 2 \wedge \neg p[i] \wedge$ *flag-zero-at-p* $\wedge\ pc'[i] = 3 \wedge$ *copy-flag*$(i) \wedge$ *copy-p*$) \vee$ $(pc[i] = 2 \wedge\ p[i] \wedge pc'[i] = 4 \wedge$ *copy-flag*$(i) \wedge$ *copy-p*$)$
$\alpha_3(i)$	$\triangleq$	$pc[i] = 3 \wedge$ *p-set*$(i) \wedge pc'[i] = 4 \wedge$ *copy-flag*(i)
$\alpha_4(i)$	$\triangleq$	$pc[i] = 4 \wedge flag'[i] = 2 \wedge pc'[i] = 5 \wedge$ *copy-p*
$\alpha_5(i)$	$\triangleq$	$(pc[i] = 5 \wedge\ \exists j \neq i flag[j] = 2 \wedge pc'[i] = 1 \wedge$ *copy-flag*$(i) \wedge$ *copy-p*$) \vee$ $(pc[i] = 5 \wedge \neg\exists j \neq i flag[j] = 2 \wedge pc'[i] = 6 \wedge$ *copy-flag*$(i) \wedge$ *copy-p*$)$
$\alpha_6(i)$	$\triangleq$	$pc[i] = 6 \wedge flag'[i] = 0 \wedge pc'[i] = 1 \wedge$ *copy-p*
$\alpha(i)$	$\triangleq$	$($*copy-other*$(i) \wedge (\alpha_1(i) \vee \alpha_2(i) \vee \alpha_4(i) \vee \alpha_5(i) \vee \alpha_6(i))) \vee$ $($*copy-other-except-p*$(i) \wedge \alpha_3(i))$
initial	$\triangleq$	$\forall i\ pc[i] = 1 \wedge flag[i] = 0 \wedge \neg p[i]$
mutex	$\triangleq$	$\Box\neg\exists i\,\exists j \neq i\ (pc[i] = 6 \wedge pc[j] = 6)$
liveness$_1$	$\triangleq$	$\Box\forall i\ ((p[i] \wedge flag[i] \neq 0 \wedge \forall j \neq i\ pc[j] \neq 3) \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{true}}$
liveness$_2$	$\triangleq$	$\Box\forall i\ ((p[i] \wedge flag[i] = 0 \wedge \forall j \neq i\ pc[j] \neq 3) \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{false}}$
liveness$_3$	$\triangleq$	$\Box\forall i\ ((p[i] \wedge pc[i] \neq 1 \wedge \forall j \neq i\ pc[j] \neq 3) \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{true}}$
liveness$_4$	$\triangleq$	$\Box\forall i\ ((p[i] \wedge \forall j \neq i\ pc[j] \neq 3) \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{false}}$
liveness$_5$	$\triangleq$	$\Box\forall i\ (p[i] \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{false}}$
liveness$_6$	$\triangleq$	$\Box\forall i\ (pc[i] = 1 \longrightarrow \Diamond\ pc[i] = 6)$ $\boxed{\text{false}}$

Figure C.9: Dijkstra's algorithm. Pseudo code for process i in $LTL(MSO)$.

Paper D

Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols (Extended Version)

The original version is to appear in LNCS, volume 4963.

Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols (Extended Version)

Mayank Saksena
mayanks@it.uu.se
Uppsala University

Oskar Wibling
oskarw@it.uu.se
Uppsala University

Bengt Jonsson
bengt@it.uu.se
Uppsala University

Abstract

We present a technique for modeling and automatic verification of network protocols, based on graph transformation. It is suitable for protocols with a potentially unbounded number of nodes, in which the structure and topology of the network is a central aspect, such as routing protocols for ad hoc networks. Safety properties are specified as a set of undesirable global configurations. We verify that there is no undesirable configuration which is reachable from an initial configuration, by means of symbolic backward reachability analysis.

In general, the reachability problem is undecidable. We implement the technique in a graph grammar analysis tool, and automatically verify several interesting nontrivial examples. Notably, we prove loop freedom for the DYMO ad hoc routing protocol. DYMO is currently on the IETF standards track, to potentially become an Internet standard.

D.1 Introduction

The verification of network protocols has been one of the most important driving forces for the development of model checking technology. Most approaches (e.g., [Hol97, CGL92]) analyze finite-state models of protocols, but an increasing number of techniques are developed for analyzing parameterized or infinite-state models (see, e.g., [AČJYK00, ZP04, AJNS04]). In this paper, we consider verification of protocols for networks with a potentially unbounded number of nodes, possibly with a

dynamically changing topology. This is a large class of protocols, including protocols for wireless ad hoc networks, many distributed algorithms, security protocols, telephone system services, etc. Global configurations of such protocols are naturally modeled using graphs, that are transformed by the dynamic behavior of the protocol, and therefore various forms of graph transformation systems have been used to model and analyze them [KK06, BBG$^+$06].

In this paper, we present a technique for modeling and verification of protocols using a variant of *graph transformation systems* (GTSs) [KK06, BBG$^+$06]. We use a general mechanism for expressing conditions on the applicability of a rule, in the form of *negative application conditions (NACs)*. Sets of global configurations are symbolically represented by *graph patterns* [BBG$^+$06], which are graphs extended with NACs. Intuitively, a graph pattern represents the set of configurations that contain it as a subgraph, but none of the NACs. A safety property of a protocol is represented by a set of graph patterns that represent undesirable global configurations.

We consider the problem of verifying safety properties. This can be reduced to the problem whether an undesirable configuration can be reached, by a sequence of graph transformation steps, from some initial global configuration. We present a method for automatically checking such a reachability problem by backward reachability analysis. Backward reachability analysis is a powerful verification technique, which has generated decidability results for many classes of parameterized and infinite-state systems (e.g., [AJ96, AČJYK00, EFM99]) and proven to be highly useful also for undecidable verification problems (e.g., [ADBR07]). By fixed-pointcomputation, we compute an over-approximation of the set of configurations from which a bad configuration can be reached, and check that this set contains no initial configuration. The central part of the backward reachability procedure is to compute the predecessors of a set of configurations in this symbolic representation. Since the reachability problem is undecidable in general, the fixed-pointcomputation is not guaranteed to terminate. However, we show that the techniques are powerful enough for verifying several interesting nontrivial examples, indicating that the approach is useful for network protocols where the dynamically changing topology of the network is a central aspect.

A main motivation for our work is to analyze protocols for wireless ad hoc networks, including the important class of *routing protocols*. We have implemented our technique, and successfully verified that the DYMO protocol [CP07a] will never generate routing loops. Verifying loop freedom for ad hoc routing protocols has been the subject of much work [BOG02, DD02]; several previous protocol proposals have been incorrect in this respect [BMJ$^+$98, AWD04]. Our verification method handles a detailed ad hoc routing protocol model, with relatively little

effort. In our work, we have also found GTSs to be an intuitive and visually clear form of modeling.

Related Work. There have been several efforts to verify loop freedom of routing protocols for ad hoc networks. Bhargavan et al. [BOG02] verified AODV [PBR99] to be loop free, using a combination of SPIN for model checking a finite network model, and HOL theorem proving for generalizing the proof. In contrast, we prove the same property automatically for an arbitrary number of nodes. Our experience is that modeling using GTSs is more intuitive than to separately construct SPIN models and HOL proofs. Das and Dill [DD02] developed automatic predicate discovery for use in predicate abstraction, and proved loop freedom for a simplified version of AODV, excluding timeouts. The construction of an abstract system and discovery of relevant abstraction predicates require many calls to a theorem prover; our method does not need to interact with a theorem prover. We check the graphs directly for inconsistencies.

There have been several other approaches to modeling and analysis using variants of GTSs. König and Kozioura [KK06] over-approximate graph transformation systems using Petri nets, successively constructed using forward counter-example guided abstraction refinement. Their technique does not support the use of NACs. We have found NACs to be an advantage during modeling and verification. For example, our first approach at verifying the DYMO protocol was without NACs, resulting in a more complex model with features not directly related to the central protocol function.

Kastenberg and Rensink [KR06] translate GTSs to finite-state models in the GROOVE tool by putting an upper bound on the number of nodes in a network. Becker et al. [BBG+06] verified safety properties of mechatronic systems, modeled by GTSs that are similar to ours. However, they only check that the set of non-bad configurations is an inductive invariant. That worked for their application, but for verifying safety properties in general it requires the user to supply an inductive invariant. Bauer and Wilhelm [BW07, Bau06] use *partner abstraction* to verify dynamic communication systems; two nodes are not distinguished if they have the same labels and the sets of labels of their adjacent nodes are equal, respecting edge directions. That abstraction is not suited for ad hoc protocols, because nodes do not have dedicated roles.

Backward reachability analysis has also been used to verify safety properties in many parameterized and infinite-state system models, with less general connection patterns than those possible in GTSs. Examples include totally homogeneous topologies in which nodes can not identify different partners, resulting in Petri nets with variants (e.g., [EFM99]), systems with linear structure and some extensions (e.g., [ADBR07]), and

systems with binary connections between nodes, tailored for modeling telephone services [JK95].

Organization of Paper. We give a brief outline of the DYMO protocol in Section D.2. The graph transformation system formalism and the backward reachability procedure are presented in Sections D.3 and D.4. In Section D.5 we describe how we modeled DYMO, and present our verification results in Section D.6. Finally, Section D.8 concludes the paper.

D.2 DYMO

We are interested in modeling and verification of ad hoc routing protocols. These protocols are used in networks that vary dynamically in size and topology. Every network node that participates in an ad hoc routing protocol acts as a router, using forwarding information stored in a routing table. The purpose of the ad hoc routing protocol is to dynamically update the routing tables so that they reflect the current network topology. DYMO [CP07a] is one of two ad hoc routing protocols currently considered for standardization by the IETF MANET group [The]. The latest DYMO version at the time of writing is specified in version 10 of the DYMO Internet draft [CP07b]. This is the version we have used as basis for our modeling. The following is a simplified description of the main properties of DYMO. The reader is referred to the Internet draft for the details.

In our protocol model, each DYMO network node A has an address, a routing table and a sequence number. The sequence number of A is included in routing messages originating from A, as a measure of freshness, and is incremented for each such message. The routing table of A contains the following fields for each destination node D.

- $\texttt{RouteNextHopAddress}_A(D)$ is the node to which A currently forwards packets, destined for node D.
- $\texttt{RouteSeqNo}_A(D)$ is the sequence number that node A has recorded for the route to destination D. It is a measure of the freshness of a route; a higher number means more recent routing information. Note that this sequence number concerns the route to D from A, and is not related to the sequence number of A.
- $\texttt{RouteHopCnt}_A(D)$ is the recorded distance from A to node D, in terms of number of hops.
- $\texttt{Broken}_A(D)$ is an indicator of whether or not the route from A to D can be used. The protocol has a mechanism to detect when

a link on a route is broken [CP07b]. Information regarding broken links is propagated through route error messages (RERR).

When a network node A wants to send a packet to another network node D, it first checks its routing table to see if it has an entry with $\texttt{Broken}_A(D) = \mathit{false}$. If that is the case, it forwards the packet to node $\texttt{RouteNextHopAddress}_A(D)$. Otherwise, node A needs to find a route to D, which it does by issuing a route request (RREQ) message. The route request is flooded through the network. It contains the addresses of nodes A and D, the sequence number of A, and a hop counter. The hop counter contains the value 1 when the RREQ is issued; each re-transmitting node then increases it by one. Node A increases its own sequence number after each issued route request.

When the destination of a route request, D, receives it, it generates a route reply message (RREP). The route reply contains the same fields as the request. Route replies are not flooded, but instead routed through the network using available routing table entries. RREPs and RREQs are collectively referred to as routing messages (RMs).

Whenever a network node A receives an RM, the routing table of A is compared to the RM. If A does not have an entry pertaining to the originator of the RM, then the information in the RM is inserted into the routing table of A. Otherwise, the information in the RM replaces that of the routing table if the information is more recent, or equally recent but better, in terms of distance to the originator. The routing table update rules are detailed in Section D.5.

D.3 Modeling Using Graph Transformation Systems

We model systems as transition systems of a particular form, in which configurations are hypergraphs, and transitions between configurations are specified by graph rewriting rules. Constraints on configurations are represented by so-called *patterns*, which are hypergraphs extended with a mechanism to describe the absence of certain hyperedges: *negative application conditions (NACs)*. Our definitions are similar to the ones used by, e.g., Becker et al. [BBG+06], but with a more general facility for expressing NACs.

Assume a finite set Λ of *labels*. A *hypergraph* is a pair $\langle N, E\rangle$, where N is a finite set of *nodes*, and $E \subseteq \Lambda \times N^*$ is a finite set of *hyperedges*. A hyperedge is a pair $(\lambda, \overrightarrow{n})$, where $\lambda \in \Lambda$ is its *label* and $\overrightarrow{n} \in N^*$. The length of $\overrightarrow{n}$ is called the *arity* of the hyperedge. A hyperedge is essentially a relation on nodes, and can be visualized as a box labeled λ, with connections to each node $n \in \overrightarrow{n}$.

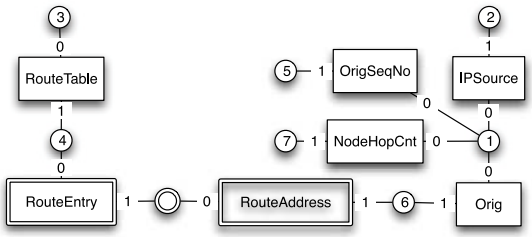


Figure D.1: A pattern containing a NAC.

A *pattern* is a tuple $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^- \rangle$, where $\langle N_\varphi, E_\varphi \rangle$ is a hypergraph, and $\mathcal{G}_\varphi^-$ is a set of NACs. Each *NAC* is a hypergraph $G^- = \langle N^-, E^- \rangle$, where N^- is a finite set of *negative nodes* disjoint from N_φ, and $E^- \subseteq \Lambda \times (N_\varphi \cup N^-)^*$ is a finite set of *negative hyperedges*. We refer to N_φ and E_φ as *positive* nodes and edges of φ. We define $\mathcal{N}(E) = \{n \in \overrightarrow{n} \mid (\lambda, \overrightarrow{n}) \in E\}$.

Example. Figure D.1 shows a pattern — the left-hand side of one of the DYMO model routing table update rules. The pattern models a network node receiving routing information for a node to which it currently has no route. In the pattern, positive nodes are drawn as circles and negative nodes as double circles. Nodes have numeric names for identification. Positive and negative edges are drawn as boxes and double boxes. Edge connections are numbered, to indicate their order. The pattern contains a single NAC, consisting of the negative edges labeled `RouteEntry` and `RouteAddress` along with their connected nodes. Without the possibility to express non-existence, we would need to model traversal through the entries to conclude the absence of an entry. In more detail, the pattern consists of a network node A (node 3) and a routing message (node 1). A has a routing table (node 4) that contains no routing table entry pointing to network node D (node 6). The message has originator D, a hop count (node 7), a sequence number (node 5) and an IP source (node 2).

A hypergraph $g = \langle N_g, E_g \rangle$ is *subsumed by* a pattern $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^- \rangle$, written $g \preceq \varphi$, if there exists an injection $h : N_\varphi \to N_g$ satisfying:

1. for each $(\lambda, \overrightarrow{n}) \in E_\varphi$ we have $(\lambda, h(\overrightarrow{n})) \in E_g$ and

2. there exists no $\langle N^-, E^- \rangle \in \mathcal{G}_\varphi^-$ and no injection $k : N^- \to N_g$ such that $(\lambda, (h \cup k)(\overrightarrow{n})) \in E_g$ for each $(\lambda, \overrightarrow{n}) \in E^-$, where $(h \cup k)$ is defined as h on N_φ and as k on N^-.

Intuitively, a pattern $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi \rangle$ is a constraint, saying that a hypergraph must contain $\langle N_\varphi, E_\varphi \rangle$ as a subgraph, which does not have a "match" for any NAC in $\mathcal{G}^-_\varphi$.

Above we let $f((n_1, \ldots, n_k)) = (f(n_1), \ldots, f(n_k))$ for a function on nodes applied to a vector of nodes. If an injection h satisfying the above conditions exists, we say that $g \preceq \varphi$ is *witnessed by* h, written $g \preceq_h \varphi$.

For a pattern φ we use $[\![\varphi]\!]$ to denote the set of hypergraphs g such that $g \preceq \varphi$. For a set of patterns Φ, we let $[\![\Phi]\!] = \cup\{[\![\varphi]\!] \mid \varphi \in \Phi\}$. We call pattern $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi \rangle$ *consistent* if $\langle N_\varphi, E_\varphi \rangle \preceq \varphi$. Informally, φ is consistent if none of its NACs contradicts its positive nodes and edges. An inconsistent pattern ψ represents an empty set, as $g \preceq \psi$ is not satisfied by any g.

A pattern φ is subsumed by the pattern ψ, denoted $\varphi \preceq \psi$, if $[\![\varphi]\!] \subseteq [\![\psi]\!]$. The relation $\preceq$ on patterns can be checked according to the following Proposition.

Proposition D.1 *Given patterns consistent patterns $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi \rangle$ and $\psi = \langle N_\psi, E_\psi, \mathcal{G}^-_\psi \rangle$, we have that $\varphi \preceq \psi$ iff there exists an injection $h : N_\psi \to N_\varphi$ such that $\langle N_\varphi, E_\varphi \rangle \preceq_h \langle N_\psi, E_\psi, \emptyset \rangle$ and for each NAC $\langle M^-, F^- \rangle \in \mathcal{G}^-_\psi$ there is a NAC $\langle N^-, E^- \rangle \in \mathcal{G}^-_\varphi$ and an injection $k : N^- \to M^-$ such that*

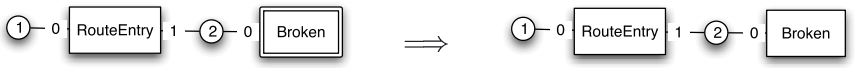
- $(\mathcal{N}(E^-) \setminus N^-) \subseteq h(N_\psi)$, *and*
- *for each $(\lambda, \overrightarrow{n}) \in E^-$, we have $(\lambda, (h^{-1} \cup k)(\overrightarrow{n})) \in F^-$.* □

Intuitively, $\varphi \preceq \psi$ if and only if the positive part of ψ is a subgraph of the positive part of φ, and for each NAC in $\mathcal{G}^-_\psi$, there is a corresponding NAC in $\mathcal{G}^-_\varphi$ which is a subgraph of the former NAC.

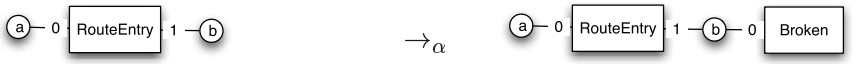
In our system model, configurations are represented by hypergraphs. Transitions are specified by *actions*, which are (hypergraph) rewrite rules.

Definition D.2 *An* action *is a pair $\langle L, R \rangle$, where $L = \langle N_L, E_L, \mathcal{G}^-_L \rangle$ is a pattern and $R = \langle N_R, E_R \rangle$ is a hypergraph with $N_L \subseteq N_R$ (i.e., actions can create nodes, but not delete them). The action $\alpha = \langle L, R \rangle$ denotes the set $[\![\alpha]\!]$ of pairs of configurations (g', g), with $g' = \langle N_{g'}, E_{g'} \rangle$, $g = \langle N_g, E_g \rangle$ and $N_{g'} \subseteq N_g$ such that there is an injection $h : N_R \to N_g$ satisfying:*

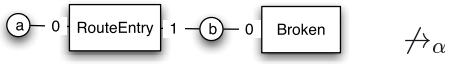
- *$g' \preceq L$ is witnessed by the restriction of h to N_L*
- *$N_g = N_{g'} \cup h(N_R)$*
- *$E_g = (E_{g'} \setminus h(E_L)) \cup h(E_R)$.* □



(a) Action α



(b) Pair of configurations $(g', g) \in [\![\alpha]\!]$



(c) Configuration g such that $\neg\exists g.(g', g) \in [\![\alpha]\!]$

Figure D.2: Example of an action and its semantics.

Example. Figure D.2(a) shows an action $\alpha = \langle L, R \rangle$. The pattern L is to the left of the arrow ($\Longrightarrow$) and R to the right. The action does not create any nodes, i.e., $N_L = N_R$. Figure D.2(b) shows a pair $(g', g) \in [\![\alpha]\!]$, i.e., g' can be rewritten via α to g. The subsumption $g' \preceq L$ is witnessed by the injection $h = \{1 \mapsto a, 2 \mapsto b\}$. The injection h satisfies $N_g = N_{g'} \cup h(N_R) = \{a, b\}$ and $E_g = (E_{g'} \setminus h(E_L)) \cup h(E_R) = h(E_R)$. Figure D.2(c) shows a configuration g' such that there is no g with $(g', g) \in [\![\alpha]\!]$, since $g' \not\preceq L$. In other words, g' cannot be rewritten via α.

Definition D.3 *A* system model *is a pair $\langle \gamma_0, \mathcal{A} \rangle$ consisting of an initial configuration γ_0 together with a finite set of actions $\mathcal{A}$.* □

For a set Γ of configurations and an action α, let $Pre(\alpha, \Gamma) = \{g' \mid \exists g \in \Gamma.\ (g', g) \in [\![\alpha]\!]\}$, i.e., the configurations which in one step can be rewritten to Γ using α. Similarly, for a set of actions $\mathcal{A}$, let $Pre^\star(\mathcal{A}, \Gamma)$ denote the set of configurations which can reach a configuration in Γ by a sequence of rewritings using actions in $\mathcal{A}$.

D.4 Symbolic Verification

We formulate a verification scenario as the problem whether a set of configurations, represented by a set of patterns, is reachable. More precisely, given a system model $\langle \gamma_0, \mathcal{A} \rangle$, and a set of patterns Φ, the *reachability problem* asks whether there is a sequence of transitions from γ_0 to some configuration in $[\![\Phi]\!]$.

In our approach, we analyze a reachability problem using *backward reachability analysis*, in which we compute an over-approximation of the set $Pre^\star(\mathcal{A}, [\![\Phi]\!])$ of configurations, and check whether it includes γ_0. We clarify why and when the computation is not exact in the *Approximation*

paragraph below. In general, the reachability problem is undecidable, and our analysis is not guaranteed to terminate. However, the technique is sufficiently powerful to verify several nontrivial network protocols (see Section D.6).

We attempt to compute $Pre^\star(\mathcal{A}, [\![\Phi]\!])$ by standard fixed-pointiteration, using predecessor computation, as shown in Procedure 1. In the proce-

Procedure 1 Backward Reachability Analysis

Require: System model $\langle \gamma_0, \mathcal{A} \rangle$ and a set Φ of (bad) patterns
Ensure: If terminates; answers whether a configuration in $[\![\Phi]\!]$ is reachable from γ_0

```
1  V := ∅, W := Φ
2  while W ≠ ∅ do
3      choose φ ∈ W
4      W := W \ {φ}
5      if γ0 ∈ [[φ]] then
6          return "Reachable"
7      if ∀ψ ∈ (V ∪ W). ¬(φ ⪯ ψ) then
8          V := V ∪ {φ}
9          for each α ∈ A do
10             W := W ∪ Pre(α, φ)
11 return "Unreachable"
```

dure, V and W are sets of patterns whose predecessors already have (V) and have not (W) been computed. In each iteration of the while loop, we choose a pattern φ from W. If $\gamma_0 \in [\![\varphi]\!]$ then we have found a (possibly spurious) path from γ_0 to $[\![\Phi]\!]$. Otherwise, we check whether φ is redundant, meaning that it is subsumed by some other pattern which will be or has been explored. If not, we add to W a set of patterns over-approximating $Pre(\mathcal{A}, [\![\varphi]\!])$. As a further optimization, not shown in Procedure 1, at line 7 we also remove patterns from V and W that are subsumed by φ; keeping V and W small speeds up the procedure.

The central part of Procedure 1 is the (nontrivial) computation of predecessors of a pattern; it is done as in Procedure 2, whose description follows. Procedure 2 terminates on any input, as all loops are finite.

Let a *partial injection*, or *matching*, from a set N to a set N' be an injection from a nonempty subset of N to N'. For two patterns $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi \rangle$ and $\psi = \langle N_\psi, E_\psi, \mathcal{G}^-_\psi \rangle$, we use $\varphi + \psi$ to denote $\langle N_\varphi \cup N_\psi, E_\varphi \cup E_\psi, \mathcal{G}^-_\varphi \cup \mathcal{G}^-_\psi \rangle$. When adding patterns, if the node and edge sets are not disjoint, the result is a "merge". No automatic renaming is assumed.

We use the following two subtraction operations in Procedure 2. First, for a pattern $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi \rangle$, and an action $\alpha = \langle L, R \rangle$, let $\varphi \ominus_\alpha R$ be the pattern $\psi = \langle N_\psi, E_\psi, \mathcal{G}^-_\varphi \rangle$, with $E_\psi = E_\varphi \setminus E_R$ and $N_\psi = N_\varphi \setminus$

Procedure 2 PRE(α, φ)

Require: Action $\alpha = \langle L, R\rangle$, pattern $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^-\rangle$
Ensure: Φ is a set of patterns satisfying $Pre(\alpha, [\![\varphi]\!]) \subseteq [\![\Phi]\!]$

```
1  Φ := ∅
2  Rename all nodes in φ, so that they are disjoint from the nodes in
   L and R
3  for each partial injection h : N_R → N_φ do
4      Rename each node h(n) in the range of h to n
5      if ∃ n ∈ Dom(h) − N_L . E_+(n, φ) ⊈ E_+(n, R) ∨
       Inconsistent(φ + R) then
6          skip
7      else
8          φ' := (φ ⊖_α R) + L
9          for each G⁻ ∈ G_φ⁻ do
10             if Inconsistent((L ⊖_E R) + G⁻) then
11                 φ' = φ' − G⁻
12         if ¬ Inconsistent(φ') then
13             Φ := Φ ∪ φ'
14 return Φ
```

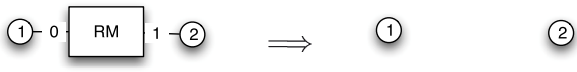
$(N_R \setminus N_L)$. Second, for a pattern $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^-\rangle$, and a hypergraph $g = \langle N_g, E_g\rangle$, let $\varphi \ominus_E g$ be the pattern $\psi = \langle N_\psi, E_\psi, \mathcal{G}_\varphi^-\rangle$, with $E_\psi = E_\varphi \setminus E_g$ and $N_\psi = \mathcal{N}(E_\psi)$.

For a NAC G^-, we use $\varphi + G^-$ to denote $\langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^- \cup G^-\rangle$ and $\varphi - G^-$ to denote $\langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^- \setminus G^-\rangle$. If $n \in N_\varphi$, let $\mathcal{E}_+(n, \langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^-\rangle)$ denote the set of edges in E_φ connected to n.

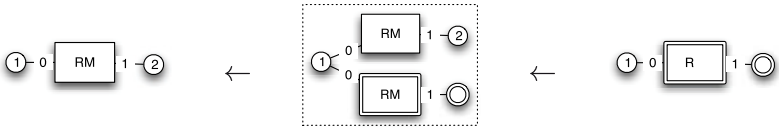
Procedure 2 first renames the nodes (line 2) to avoid unintended node collisions between φ and α. Thereafter, the loop starting at line 3 performs a sequence of operations for each possible matching between some nodes of N_R and N_φ.

On line 4 each node $h(n)$ in the range of h is renamed to n, in order to "merge" R and φ according to h. Since nodes that are created by α must also have all their edges created by α, we should discard matchings which violate this (line 5). On line 5 we also discard inconsistent matchings. The check $\mathcal{I}nconsistent(\varphi)$ is true iff pattern φ is not consistent.

On line 8 the action α is "executed" backwards to obtain a pattern φ' that is a potential predecessor of φ. Using the special subtraction $\ominus_\alpha$ nodes and edges created by α are removed from φ. On lines 9–11, we remove all NACs from φ' which contradict subgraphs removed by α. The backward execution (on line 8) and the NAC removal may both introduce approximation (see the paragraph below). Since by definition α cannot remove nodes, we use the special subtraction $\ominus_E$ which ignores nodes not connected to edges. On line 12, we discard the resulting predecessor



(a) Idealized action α



(b) Predecessor computation showing intermediate pattern

Figure D.3: Approximation due to upwards-closure.

pattern if it is inconsistent — this can happen if a NAC in L contradicts a positive subgraph of φ'. Finally, if we reach line 13, we have found a predecessor pattern, which is added to Φ.

Approximation. The predecessor computation in Procedure 2 may introduce an approximation at line 8 or line 11. If α removes a hyperedge $(\lambda, \overrightarrow{n})$ of arity zero or one, and this edge is included in φ, then $Pre(\alpha, [\![\varphi]\!])$ should contain two copies of $(\lambda, \overrightarrow{n})$, representing "two or more" occurrences of $(\lambda, \overrightarrow{n})$. However, the use of sets to contain the hyperedges of patterns results in $\textsc{Pre}(\alpha, \varphi)$ containing only one copy of $(\lambda, \overrightarrow{n})$, after line 8, representing "one or more" occurrences of $(\lambda, \overrightarrow{n})$.

Further, if α removes a subgraph which is forbidden by φ, then $Pre(\alpha, [\![\varphi]\!])$ should say that there is exactly one subgraph of this form. However, patterns cannot always express "exactly one" occurrence of a subgraph. In this situation, Procedure 2 therefore lets the resulting pattern say that "there is at least one occurrence" of this subgraph. As an example, consider the simple situation in Figure D.3, where α, shown in Figure D.3(a), removes an `RM`-edge between two nodes, and φ, the rightmost pattern in Figure D.3(b), says that there is no `RM`-edge. The exact predecessor of φ is: "there is exactly one `RM`-edge between two nodes". However, the resulting predecessor (the leftmost pattern in Figure D.3(b)) represents that there is *at least* one `RM`-edge connected to graph node 1. To illustrate the effect of lines 9–11 of Procedure 2, an intermediate pattern, where the contradiction has not yet been resolved, is shown in Figure D.3(b).

Optimizations. To make the analysis more efficient, we have (implemented) two mechanisms for the user to specify simple type constraints. One is to annotate nodes with types that are respected in the analysis, with the semantics that nodes may only "match" nodes of same type. Another is to add patterns that describe multiplicity constraints

on edges. For example, our DYMO models use "a network node can have at most one routing table", by specifying a pattern where a node has two routing tables as "impossible".

We need to model integer-valued variables, as DYMO uses sequence numbers and hop counts. This is done by representing integers as nodes, and greater than ($>$) and equality ($=$) relations as edges between these nodes. We do not represent concrete integer values. Hence, we cannot compare integers which are not connected by a relational edge. We have extended our tool to handle the transitive closure of $>$ and $=$, as part of the predecessor computation. For each predecessor pattern generated, the closure of all transitive numerical relations present in the pattern is computed. New relational edges are then added to the pattern accordingly. The reason is that our syntactic subsumption check cannot deduce such semantic information about relations. The check for created nodes on line 5 of Procedure 2 was also extended to take into account the transitivity of numerical relations.

D.5 Modeling and Verification of DYMO

In this section we describe how we modeled the DYMO protocol (more precisely, the latest version at the time of writing, version 10 [CP07b], and version 5). See our project home page [GBT] for the complete models. In total, our DYMO v10 model consists of one initial graph ("an empty network") and 77 actions. Of these, 38 actions model routing table update rules, similar to the one in Figure D.4 below. We have only used unary and binary hyperedges in our models, although our implementation supports hyperedges of any arity.

Modeling Network Topology and Message Transmission. We represent arbitrary network topologies by letting the initial system configuration be an empty network (i.e., an empty graph), and including an action for creating an arbitrary network node; thus any initial topology can be formed. We do not explicitly model connectivity in the network. Instead all nodes can potentially react on all messages in the network; this reaction on a message can be postponed indefinitely, corresponding to a node being out of range or otherwise incapable of receiving the message. Messages can also be non-deterministically removed, corresponding to message loss. In our modeling of message transmission, messages are left in the network after a node has handled them (until they are potentially dropped): this accounts for messages being duplicated.

Handling of Timeouts and Hop Limits. DYMO uses timeouts to determine if a RREQ should be retransmitted, if a link is broken, or

if a routing table entry should be removed. We over-approximate timeouts as "event x can happen at any time", which covers all possibilities for a timeout. It is known from previous work on the AODV protocol [BOG02], that if entries are removed from the routing table, loops may form. The reason is that obsolete information can then be accepted. In DYMO, routing table entries are invalidated (set to broken) after some time, and later removed; temporary loops are thus tolerated. We exclude removal of routing table entries from our analysis; they can only be invalidated. In practice, we thus verify loop-freedom under the assumption that routing table entries are kept "long enough".

We do not model DYMO hop limits [CP07b], used to limit packet traversal. However, since we include actions for arbitrary dropping of RMs and RERRs, we implicitly cover all possible hop limit settings.

Routing Table Update Rules. The DYMO specification [CP07b] prescribes when a node should update its own routing table upon receiving routing data, i.e., when received routing data should replace existing data. Existing data is represented by a routing table entry, with fields `RouteSeqNo`, `RouteHopCnt`, and `Broken`. Received data is represented by a routing message with fields `OrigSeqNo`, `NodeHopCnt` and message type `RM` – either a route request (`RREQ`) or a route reply (`RREP`). The table entry should be updated in the following cases:

1. $\texttt{OrigSeqNo} > \texttt{RouteSeqNo}$
2. $\texttt{OrigSeqNo} = \texttt{RouteSeqNo} \wedge \texttt{NodeHopCnt} < \texttt{RouteHopCnt}$
3. $\texttt{OrigSeqNo} = \texttt{RouteSeqNo} \wedge \texttt{NodeHopCnt} = \texttt{RouteHopCnt} \wedge \texttt{RM} = \texttt{RREP}$
4. $\texttt{OrigSeqNo} = \texttt{RouteSeqNo} \wedge \texttt{NodeHopCnt} = \texttt{RouteHopCnt} \wedge \texttt{Broken}$

The rules say that an update is allowed if (1) the message has a higher sequence number for the destination, or (2) the message has the same sequence number, but a shorter route, or (3) the message has the same routing metric value, and the message is a route reply, or (4) the table entry is broken. See Figure D.4 for an illustration of how we model the update rules. The figure corresponds to rule (2). In our framework, we have to model each combination of network nodes used in the rules, such as when `IPSource` equals `Orig`, or `RouteNextHopAddress` equals `RouteAddress`, etc., as separate actions; however, we have tool support for doing this.

Formalizing the Non-Looping Property. A central property of ad hoc routing protocols is that they never cause routing loops, as a routing

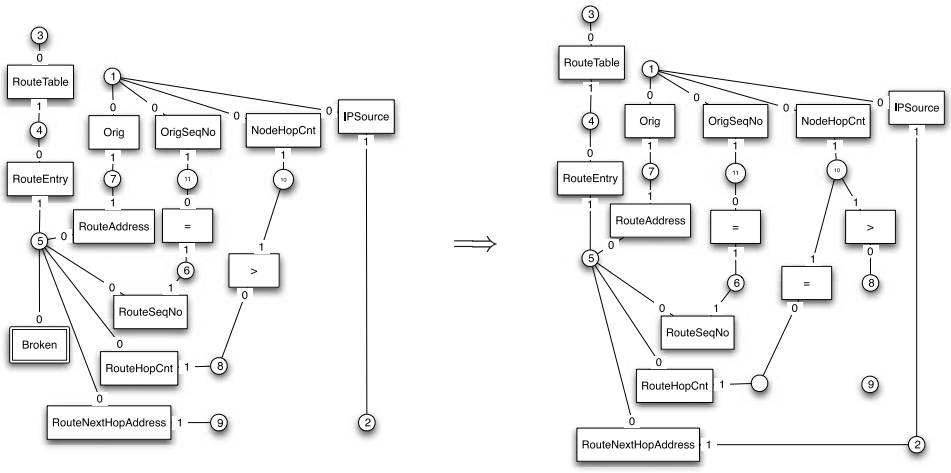


Figure D.4: Action modeling a routing table update.

loop prevents a packet from reaching its intended destination. A routing loop is a nonempty finite sequence of nodes $n_1, \ldots, n_k$ such that for some destination D it holds that for all $i : 1 \leq i \leq k$ node $n_{(i+1)(mod\ k)}$ is the next hop towards D from node n_i , and $n_i \neq D$.

We define the ordering $<_D$ on nodes in a configuration as: $n <_D n'$ iff $\texttt{RouteSeqNo}_n(D) > \texttt{RouteSeqNo}_{n'}(D) \vee (\texttt{RouteSeqNo}_n(D) = \texttt{RouteSeqNo}_{n'}(D) \wedge \texttt{RouteHopCnt}_n(D) < \texttt{RouteHopCnt}_{n'}(D))$. There can be no routing loops towards a destination D, if each hop from a node n towards D goes to a node n' with $n' <_D n$. Since $<_D$ is a partial order, any routing path towards D can contain a node at most once. The same ordering was used in the proof of loop freedom for AODV in [BOG02]. The following property, LP, implies the pairwise ordering along routing paths; if LP is invariant for DYMO, there are no routing loops.

$$\forall\, A, B, D \quad A \neq B, B \neq D, A \neq D$$
$$\texttt{RouteNextHopAddress}_A(D) = B \implies B <_D A \qquad (LP)$$

By negating the loop property (LP), we obtain a characterization of the bad system configurations. Loops may thus form if the sequence number strictly decreases, or the sequence number stays the same but the hop count does not decrease, between a node A and its next hop B on a route towards a destination node D. In our verification of DYMO, we verify unreachability for a set of six bad patterns. Three represent a disjunct of $(\neg LP)$ under quantification; two represent a network node with a routing table entry pointing to the node itself; and one pattern represents that a node has a next hop (which is not D) towards some

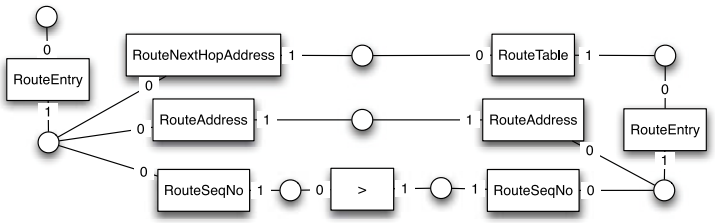


Figure D.5: Graph pattern representing a set of bad system configurations in DYMO.

destination D, but the next hop has no entry for D. As an example, a pattern representing one of the disjuncts of $(\neg LP)$ is shown in Figure D.5.

D.6 Experimental Results

We have modeled and verified the DYMO protocol as described in Sections D.5 and D.4. Recall that the analysis is under an assumption of routing table entries not being removed. The analysis has been performed using our tool GBT (Graph Backwards Tool). GBT and the models are available at our project home page [GBT]. The tool uses the `.dot` format for describing hypergraphs and patterns (input and output). If the initial configuration can be reached, an error trace, showing a sequence of actions leading to one of the bad patterns, is provided. Note that this trace may be spurious, due to over-approximation.

We have verified the latest DYMO version at the time of writing, namely version 10 of the Internet draft [CP07b], as well as an older draft (version 5). Our results are presented in Table D.1. In the "dest. reply" models, only the destination node replies to an RREQ, whereas in "interm. reply", intermediate nodes may also reply (in case they have a fresh enough route, see [CP07b]). Column $\mathcal{A}$ contains the number of actions in the model. *Seen* contains the total number of unique non-impossible patterns generated by the predecessor computation, plus the ones given as input. $\preceq$ contains the patterns which were subsumed (see Section D.4). *Left* contains the patterns left after the analysis has finished; none of them contain the initial graph. *Time* contains the total verification time (GBT start to end) on a machine with an AMD Opteron 2220 2.8 GHz processor.

In Table D.1 we have also included GBT verification results for the "Public/private servers" and "Firewall" examples, used by König and Kozioura [KK06]. These examples required modifications to work with our tool. The abstraction introduced from using sets to contain the hyperedges of patterns required us to add a zero arity edge to the right

Table D.1: *Measurement results from using GBT. Updated results are available in [SWJ07].*

Protocol	$\mathcal{A}$	**Seen**	$\preceq$	**Left**	**Time**
DYMO draft 10					
- dest. reply	56	185751	185695	56	2h 24 min
- interm. reply	77	295164	295108	56	4h 31 min
DYMO draft 05	50	118685	118637	48	1h 20 min
Pub/priv srv I	12	498	484	14	0.73 s
Pub/priv srv II	13	629	609	20	0.94 s
Firewall I	6	129	126	3	0.11 s
Firewall II	6	129	126	3	0.11 s

hand side of two actions in "Public/private servers II". The transitivity handling in our tool was also extended to include communication channels.

D.7 Proofs

We prove that our backward reachability analysis (Procedure 1) is correct.

In practice, the analysis has to use the syntactic characterization of $\preceq$ to check subsumption of patterns, as stated in Proposition D.1. Thus, we will have to prove Proposition D.1 .

The underlying assumption in Procedure 1 is that we can throw away patterns φ which are subsumed by some previously seen pattern ψ (see line 7). The motivation is that, since any hypergraph in $[\![\varphi]\!]$ is contained in $[\![\psi]\!]$, it should suffice to take predecessors of ψ . We prove that, indeed, our predecessor computation on patterns (Procedure 2) is adequate — meaning that $\textsc{Pre}(\alpha, \varphi)$ subsumes $\mathit{Pre}\,(\alpha, [\![\varphi]\!])$.

Notations and Conventions We will use the following notations and conventions in the proofs.

If an injection h satisfying the conditions of Proposition D.1 exists for patterns φ and ψ , we say that $\varphi \preceq \psi$ is *witnessed by* h, written $\varphi \preceq_h \psi$.

An injection $h : N \to N'$ can be applied on any pattern, hypergraph, edge or NAC in a distributed fashion, with the convention that $h(n) = n$ for nodes $n \notin N$. For example, $h(\mathcal{G}^-_\varphi)$ is the set $\{h(G^-_\varphi) \mid G^-_\varphi \in \mathcal{G}^-_\varphi\}$ etc.

For a set of NACs $\mathcal{G}^-_\varphi$ we let $\mathcal{N}(\mathcal{G}^-_\varphi) = \{\mathcal{N}(E^-) \mid \langle N^-, E^-\rangle \in \mathcal{G}^-\}$.

Two sets of NACs $\mathcal{G}^-_\varphi$ and $\mathcal{G}^-_\psi$ are *isomorphic*, written $\mathcal{G}^-_\varphi \simeq \mathcal{G}^-_\psi$, if there exists a bijection $h : \mathcal{N}(\mathcal{G}^-_\psi) \to \mathcal{N}(\mathcal{G}^-_\varphi)$ such that $h(\mathcal{G}^-_\psi) = \mathcal{G}^-_\varphi$ and $h^{-1}(\mathcal{G}^-_\varphi) = \mathcal{G}^-_\psi$.

Let $\mathcal{N}$ be the set of all nodes (negative and positive).

The *identity* between sets of nodes is the bijection $Id : \mathcal{N} \to \mathcal{N}\,;\, h(n) = n$.

D.7.1 Pattern Subsumption

Proposition D.1. *Given patterns* $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi\rangle$ *and* $\psi = \langle N_\psi, E_\psi, \mathcal{G}^-_\psi\rangle$ *which are consistent, we have that* $\varphi \preceq \psi$ *iff there exists an injection* $h : N_\psi \to N_\varphi$*, such that (1)* $\langle N_\varphi, E_\varphi\rangle \preceq_h \langle N_\psi, E_\psi, \emptyset\rangle$ *and (2) for each NAC* $\langle M^-, F^-\rangle \in \mathcal{G}^-_\psi$ *there is a NAC* $\langle N^-, E^-\rangle \in \mathcal{G}^-_\varphi$ *and an injection* $k : N^- \to M^-$ *such that*

- $(\mathcal{N}(E^-) \setminus N^-) \subseteq h(N_\psi)$*, and*
- *for each* $(\lambda, \overrightarrow{n}) \in E^-$*, we have* $(\lambda, (h^{-1} \cup k)(\overrightarrow{n})) \in F^-$.

Proof. ("$\Longrightarrow$") Assume that $\varphi \preceq \psi$, i.e., $[\![\varphi]\!] \subseteq [\![\psi]\!]$.

Since φ is consistent, we know that $\langle N_\varphi, E_\varphi\rangle \in [\![\varphi]\!]$, and by assumption, $\langle N_\varphi, E_\varphi\rangle \in [\![\psi]\!]$, so $\langle N_\varphi, E_\varphi\rangle \preceq \psi$. There thus exists h witnessing $\langle N_\varphi, E_\varphi\rangle \preceq \langle N_\psi, E_\psi, \emptyset\rangle$.

Now suppose that every h witnessing $\langle N_\varphi, E_\varphi\rangle \preceq \langle N_\psi, E_\psi, \emptyset\rangle$ fails to satisfy the condition (2). Thus, for every h there exists $G^-_h \in \mathcal{G}^-_\psi$ such that for each $G^-_\varphi \in \mathcal{G}^-_\varphi$ there is no k with $(h^{-1} \cup k)(G^-_\varphi) \subseteq G^-_h$.

Let $H = \{h \mid \langle N_\varphi, E_\varphi\rangle \preceq_h \langle N_\psi, E_\psi, \emptyset\rangle\}$, which contains at least one element, as shown above. For each $h \in H$, we define a hypergraph G^+_h, where $G^+_h = \langle N^+_h, E^+_h\rangle$. Intuitively, G^+_h is a *positive interpretation* of $G^-_h = \langle N^-_h, E^-_h\rangle$ where the negative edges are interpreted as positive edges with the same labels. More precisely, $N^+_h = h^+(N^-_h) \cup (h \cup h^+)(\mathcal{N}(E^-_h))$ and $E^+_h = (h \cup h^+)(E^-_h)$ with the injection $h^+ : N^-_h \to N^+_h$ and $(N^+_h \cap N_\varphi) = \emptyset$.

Now consider the hypergraph $g_l = \langle N_\varphi, E_\varphi\rangle \cup \bigcup\{G^+_h \mid h \in H\}$. Intuitively, g_l consists of the positive part of φ and for each $h \in H$ a part which contradicts a NAC in ψ but not a NAC in φ.

Now note that $g_l \preceq \varphi$. In fact, $g_l \preceq_h \varphi$ for any $h \in H$ since we do not contradict $\mathcal{G}^-_\varphi$ by our assumption. Moreover, no matter which $h \in H$ we choose, we get $\neg(g_l \preceq_h \psi)$, since we map to a part which contradicts $G^-_h \in \mathcal{G}^-_\psi$. But H contains all possible witnesses of $\langle N_\varphi, E_\varphi\rangle \preceq_h \langle N_\psi, E_\psi, \emptyset\rangle$. Hence, $g_l \in [\![\varphi]\!] \setminus [\![\psi]\!]$, contradicting $[\![\varphi]\!] \subseteq [\![\psi]\!]$. Therefore, there must exist an h satisfying (1) which also satisfies (2).

("$\Longleftarrow$") Given an injection $h : N_\psi \to N_\varphi$ and a set of injections $k_{G^-} : N^- \to M^-$ — one for each NAC $\langle M^-, F^-\rangle \in \mathcal{G}^-_\psi$ as defined above — satisfying (1) and (2). We show that $[\![\varphi]\!] \subseteq [\![\psi]\!]$.

Consider any $g = \langle N_g, E_g \rangle \in [\![\varphi]\!]$. There exists an injection $h_g : N_\varphi \to N_g$ witnessing $g \preceq \varphi$. Consider also the *composed* injection $h' = h_g \circ h : N_\psi \to N_g$; $n \mapsto h_g(h(n))$. It witnesses $g \preceq \psi$ as we will see.

We have $h'(E_\psi) \subseteq E_g$ since for each $(\lambda, \overrightarrow{n}) \in E_\psi$ there is a corresponding edge $h((\lambda, \overrightarrow{n})) \in E_\varphi$ and again $h_g(h((\lambda, \overrightarrow{n}))) \in E_g$.

Now suppose that $g \not\preceq \psi$ because the second condition (on page 6) fails. Then there exists some NAC $G^- = \langle M^-, F^- \rangle \in \mathcal{G}^-_\psi$ which contradicts g. More precisely, there exists an injection $k' : M^- \to N_g$ such that for each $(\lambda, \overrightarrow{n}) \in F^-$ we have $(\lambda, (h' \cup k')(\overrightarrow{n})) \in E_g$.

But then there is a corresponding NAC $\langle N^-, E^- \rangle \in \mathcal{G}^-_\varphi$ which also contradicts g. To see this, we use the injection $k_{G^-} : N^- \to M^-$ with the properties defined above, and the injection $k'' = k' \circ k_{G^-} : N^- \to N_g$; $n \mapsto k'(k_{G^-}(n))$. Now for each $(\lambda, \overrightarrow{n}) \in E^-$ we have $(\lambda, ((h' \circ h^{-1}) \cup k'')(\overrightarrow{n})) \in E_g$. But this contradicts $g \preceq \varphi$. Hence, $g \preceq \psi$ and $\varphi \preceq \psi$.

□

D.7.2 Correctness of the Predecessor Calculation

We prove the correctness of Procedure 2. First we establish a property of the symbolic predecessor computation, which will be used in the proof.

Symbolic Predecessor Computation

We prove a useful property of the symbolic predecessor computation.

Proposition D.4 *Given patterns $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi \rangle$, $\psi = \langle N_\psi, E_\psi, \mathcal{G}^-_\psi \rangle$ and an action $\alpha = \langle L, R \rangle$. If $\varphi \preceq_h \psi$ and $h(\mathcal{G}^-_\psi) \simeq \mathcal{G}^-_\varphi$, then for each $\varphi' \in \textsc{Pre}(\alpha, \varphi)$ there exists some $\psi' \in \textsc{Pre}(\alpha, \psi)$ such that $\varphi' \preceq \psi'$.*

Proof. Assume that $\varphi \preceq_h \psi$ and $h(\mathcal{G}^-_\psi) \simeq \mathcal{G}^-_\varphi$. We show that for each predecessor φ' of φ there exists a predecessor ψ' of ψ such that $\varphi' \preceq \psi'$ as illustrated below.

$$\begin{array}{ccc} \varphi' & \xleftarrow{\textsc{Pre}(\alpha)} & \varphi \\ \rotatebox{90}{$\preceq$} & & \rotatebox{90}{$\preceq_h$} \\ \exists\, \psi' & \xleftarrow{\textsc{Pre}(\alpha)} & \psi \end{array}$$

After renaming the nodes so that the sets of nodes are disjoint, we compute $\textsc{Pre}(\alpha, \varphi)$ and $\textsc{Pre}(\alpha, \psi)$ according to Procedure 2.

We consider all predecessors of φ. Let thus $h_{R\varphi} : N_R \to N_\varphi$ be the chosen partial injection on line 3 which causes the predecessor φ' of φ to

be output. We show that a corresponding injection gives us the desired predecessor ψ' — namely the following:

$$h_{R\psi} = h^{-1} \circ h_{R\varphi} : N_R \to N_\psi \,;\, n \mapsto h^{-1}(h_{R\varphi}(n)) \,.$$

We now argue line by line, of Procedure 2, that subsumption is preserved during the predecessor computation for our choices of injections. Initially, we have $\varphi \preceq_h \psi$. Clearly, after the renaming done on line 4 we still have subsumption. The following gives us a simpler correspondence between φ and ψ.

Lemma D.5 *If $\varphi \preceq_h \psi$ then $\widehat{\varphi} \preceq_{Id} \widehat{\psi}$, where $\widehat{\varphi} = h^{-1}_{R\varphi}(\varphi)$ and $\widehat{\psi} = h^{-1}_{R\psi}(\psi)$ are the patterns obtained after the renaming.*

Proof. *Since $\varphi \preceq_h \psi$ we have $h^{-1}(\varphi) \preceq_{Id} \psi$. Since the sets N_ψ and N_R are disjoint, we can apply $h^{-1}_{R\psi}$ to both sides, obtaining $h^{-1}_{R\psi} \circ h^{-1}(\varphi) \preceq_{Id} \widehat{\psi}$. By definition $h^{-1}_{R\psi} = (h^{-1} \circ h_{R\varphi})^{-1} = (h^{-1}_{R\varphi} \circ h)$ so the left hand side becomes $\widehat{\varphi}$.*

□

For readability, we abuse notation slightly, and continue to call the renamed patterns φ and ψ. Let thus $\varphi := h^{-1}_{R\varphi}(\varphi)$ and $\psi := h^{-1}_{R\psi}(\psi)$. By Lemma D.5 we thus have $\varphi \preceq_{Id} \psi$.

We continue with the test on line 5. It suffices to show that if φ is not skipped, i.e. both clauses are false, then ψ is not skipped either.

Lemma D.6 *If the first clause of the test on line 5 is false for φ then it is also false for ψ.*

Proof. *That the first clause is false, means that for the quantified n, $\mathcal{E}_+(n, \varphi) \subseteq \mathcal{E}_+(n, R)$. Since $\varphi \preceq_{Id} \psi$ we get $N_\psi \subseteq N_\varphi$ and $E_\psi \subseteq E_\varphi$, and the statement follows.*

□

Lemma D.7 *If the second clause of the test on line 5 is false for φ then it is also false for ψ.*

Proof. *That the second clause is false for φ means that $\neg\, \mathcal{I}nconsistent(\varphi + R)$. This means that there is no NAC in $\mathcal{G}^-_\varphi$ which contradicts R. The statement follows since $\varphi \preceq_{Id} \psi$ as for Lemma D.6.* □

Next we show that the computation on line 8 preserves subsumption. Let $\varphi' = (\varphi \ominus_\alpha R) + L$ and $\psi' = (\psi \ominus_\alpha R) + L$.

Lemma D.8 *If $\varphi \preceq_{Id} \psi$ then $\varphi' \preceq_{Id} \psi'$.*

Proof. *Suppose that* $\varphi \preceq_{Id} \psi$.

$$(\varphi \ominus_\alpha R) + L = \langle N_\varphi \setminus (N_R \setminus N_L) \cup N_L, E_\varphi \setminus E_R \cup E_L, \mathcal{G}^-_\varphi \cup \mathcal{G}^-_L \rangle$$

$$(\psi \ominus_\alpha R) + L = \langle N_\psi \setminus (N_R \setminus N_L) \cup N_L, E_\psi \setminus E_R \cup E_L, \mathcal{G}^-_\psi \cup \mathcal{G}^-_L \rangle$$

We check the conditions of Proposition 1 after line 8.

1. *This condition is true, because the same edges are removed from and added to* φ *and* ψ .

2. *This condition holds, because the same NACs are added to* φ *and* ψ .

□

We show that the NAC handling done on lines 9–11 also preserves subsumption. We will use φ'' and ψ'' to denote the results from executing lines 9–11.

Lemma D.9 *If* $\varphi' \preceq_{Id} \psi'$ *then* $\varphi'' \preceq_{Id} \psi''$.

Proof. *Because the sets of NACs are isomorphic, whenever a NAC is removed from* φ' *it is also removed from* ψ' *. Hence subsumption is preserved.* □

For the inconsistency check on line 12 of Procedure 2 we conclude that since we have $\varphi'' \preceq_{Id} \psi''$, if ψ'' is inconsistent, then so is φ'' . Hence, if φ'' passes the test, it is not inconsistent and neither is ψ'' .

Let us return to the original notations, used in the statement. We simply rename our patterns: $\varphi' := \varphi''$ and $\psi' := \psi''$. Since φ' is a predecessor of φ , it will pass this last inconsistency check. It follows that so will ψ' . Hence, we have found our ψ' with $\psi' \in \textsc{Pre}(\alpha, \psi)$ such that $\varphi' \preceq \psi'$, concluding the proof. □

Main Proof

Now we continue with the main proof. We will use the following correspondence between graph and pattern subsumption.

Lemma D.10 *Given a hypergraph* $g = \langle N_g, E_g \rangle$ *and a consistent pattern* $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}^-_\varphi \rangle$.

$$g \preceq_h \varphi \quad \Longleftrightarrow \quad \varphi_g = \langle N_g, E_g, h(\mathcal{G}^-_\varphi) \rangle \preceq_h \varphi \text{ and } \varphi_g \text{ is consistent .}$$

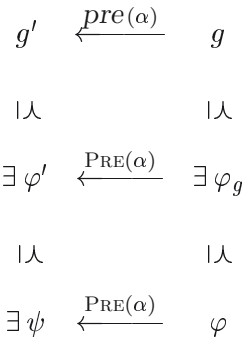


Figure D.1: Proof strategy. We show that the patterns preceded by "∃" exist. The proposition statement then follows by transitivity of $\preceq$.

Proof. *($\Longrightarrow$). Suppose $g \preceq_h \varphi$. Then g does not contradict any NAC of φ. Hence, $\langle N_g, E_g, h(\mathcal{G}_\varphi^-)\rangle$ is consistent and subsumed by φ (witnessed by h and the identity mapping between the NACs).*

($\Longleftarrow$). Suppose $\varphi_g = \langle N_g, E_g, h(\mathcal{G}_\varphi^-)\rangle \preceq_h \varphi$ and φ_g is consistent. Since φ_g is consistent, we get $g = \langle N_g, E_g\rangle \preceq_{Id} \varphi_g$. Furthermore, since $\varphi_g \preceq \varphi$ we have $g \preceq_{Id} \varphi_g \preceq \varphi$. Hence, $g \preceq \varphi$. □

Finally, we are ready to prove correctness.

Proposition D.11 *Given an action $\alpha = \langle L, R\rangle$, and a consistent pattern $\varphi = \langle N_\varphi, E_\varphi, \mathcal{G}_\varphi^-\rangle$.*

$$\Phi = \textsc{Pre}(\alpha, \varphi) \text{ satisfies } Pre(\alpha, [\![\varphi]\!]) \subseteq [\![\Phi]\!]\,.$$

Proof. Our proof strategy is depicted in Figure D.1. We consider any pair of graphs $(g',\, g) \in [\![\alpha]\!]$ where $g \preceq \varphi$. We first show that there exist patterns φ_g and φ' as shown in the figure — i.e., such that $g \preceq \varphi_g$, $\varphi_g \preceq \varphi$, $\varphi' \in \textsc{Pre}(\alpha, \varphi_g)$ and $g' \preceq \varphi'$. Once this has been established, we get by Proposition D.4 that there exists a pattern $\psi \in \textsc{Pre}(\alpha, \varphi)$ such that $\varphi' \preceq \psi$. Finally, we get $g' \preceq \psi$ by transitivity.

By Lemma D.10 we get that $\varphi_g = \langle N_g, E_g, \mathcal{G}_\varphi^-\rangle \preceq \varphi$ and, by consistency, that $g \preceq \varphi_g$. Now, by Proposition D.4, we get that for any $\varphi' \in \textsc{Pre}(\alpha, \varphi_g)$ there exists $\psi \in \textsc{Pre}(\alpha, \varphi)$ with $\varphi' \preceq \psi$. It now suffices to show that there exists a predecessor φ' of φ_g such that $g' \preceq \varphi'$.

Suppose that the injection $h : N_R \to N_g$ relates g' and g according to Definition D.2. Hence we have $g = \langle N_g, E_g\rangle$ with $N_g = N_{g'} \cup h(N_R)$ and $E_g = E_{g'} \backslash h(E_L) \cup h(E_R)$. We show that, if we choose this *same* injection on line 3 in the computation of $\textsc{Pre}(\alpha, \varphi)$, we obtain an adequate φ'. Let us, then, go through lines 4–13 of Procedure 2 using the injection $h : N_R \to N_g$ from above.

- *Line 4.* After the renaming we obtain $\widehat{\varphi_g} = \langle h^{-1}(N_{g'}) \cup N_R\,,\, h^{-1}(E_{g'}) \setminus E_L \cup E_R\,,\, h^{-1}(\mathcal{G}^-_\varphi)\rangle\,.$

- *Line 5, first clause.* This clause is false, since the edges of $\widehat{\varphi_g}$ are $h^{-1}(E_{g'}) \setminus E_L \cup E_R$. Thus $\mathcal{E}_+(n, \widehat{\varphi_g})$ is clearly a subset of $\mathcal{E}_+(n, E_R)$ for $n \in (N_R \setminus N_L)$ (in fact, the sets are equal).

- *Line 5, second clause.*

$$\widehat{\varphi_g} + R =$$

$$\langle h^{-1}(N_{g'}) \cup N_R \cup N_R,\, h^{-1}(E_{g'}) \setminus E_L \cup E_R \cup E_R,\, h^{-1}(\mathcal{G}^-_\varphi)\rangle = \widehat{\varphi_g}$$

  Since $\widehat{\varphi_g}$ is consistent, so is $\widehat{\varphi_g} + R$.

- *Line 8.* We here obtain the pattern

$$\varphi' = (\widehat{\varphi_g} \ominus_\alpha R) + L = \langle h^{-1}(N_{g'})\,,\, h^{-1}(E_{g'})\,,\, h^{-1}(\mathcal{G}^-_\varphi) \cup \mathcal{G}^-_L\rangle\,.$$

- *Lines 9–11.* Suppose that $g' \not\preceq \varphi'$. Since condition 1 of subsumption is satisfied (as defined on page 6), the reason must be that condition 2 is not. But since by definition $g' \preceq_h L$, the violated NAC must be in $\mathcal{G}^-_\varphi$. Moreover, since $\varphi + R$ is consistent, the part of φ' which contradicts the NAC cannot be in R . The only remaining alternative is that something (positive) in $L \ominus_E R$ contradicts the NAC. Hence, condition 2 is met, and $g' \preceq \varphi'$, if all contradictions of this form are resolved. This is precisely what is done on lines 9–11.

- *Line 12.* Finally, φ' is consistent, since $g' \preceq \varphi'$, and the test on this line is passed. □

D.8 Conclusions and Future Work

We have described and implemented a general framework for modeling and verification of protocols using a variant of graph transformation systems, and applied it to automatically prove loop freedom of the DYMO v10 ad hoc routing protocol. We expect that several of the actions used in our DYMO model need only small modifications to work for other ad hoc routing protocols categorized as reactive (i.e., on-demand). The reason is that reactive ad hoc routing protocols generally use the same kind of flooding route discovery mechanism; examples include AODV[PBR99], DSR[JMB01], and LUNAR[TGRW04] (see [Lis] for an extensive list).

As GTSs with NACs make up quite a generic modeling framework, there should be possibilities for interesting case studies, and further development. Directions for future work include further optimizations of

the predecessor computation, e.g., by early detection of unfruitful matchings. We are currently working on a new DYMO model, to investigate the effect on run-time performance when using hyperedges of arity greater than two. Termination of the reachability analysis can be obtained by bounding and truncating the generated patterns, at the cost of over-approximation, e.g., by enforcing a maximum size. The possibility of spurious counter-examples, due to approximations in the predecessor computation, motivates looking at counter-example guided abstraction refinement.

Acknowledgments.

We would like to thank Barbara König for valuable help on the Augur tool and related issues. We also thank Parosh Abdulla, Joachim Parrow, and the anonymous referees for their many helpful comments.

D.9 Bibliography

[AČJYK00] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[ADBR07] P.A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Regular model checking without transducers. In *Proc. TACAS '07,* 13^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[AJNS04] P.A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *Proc. CONCUR 2004,* 15^{th} *Int. Conf. on Concurrency Theory*. Springer Verlag, 2004.

[AWD04] Mehran Abolhasan, Tadeusz Wysocki, and Eryk Dutkiewicz. A review of routing protocols for mobile ad hoc networks. *Ad Hoc Networks*, 2(1):1–22, January 2004.

[Bau06] Jörg Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Universität des Saarlandes, 2006.

[BBG+06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *Proc. ICSE '06,* 28^{th} *Int. Conf. on Software Engineering*, pages 72–81. ACM Press, 2006.

[BMJ+98] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of MobiCom'98*, October 1998.

[BOG02] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538–576, 2002.

[BW07] Jörg Bauer and Reinhard Wilhelm. Static Analysis of Dynamic Communication Systems. In *14th International Static Analysis Symposium*. Springer, 2007.

[CGL92] E. M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proc.* 19^{th} *ACM Symp. on Principles of Programming Languages*, 1992.

[CP07a] Ian D. Chakeres and Charles E. Perkins. DYMO - Dynamic MANET On-demand Routing Protocol home page. `http://www.ianchak.com/dymo/`, May 2007.

[CP07b] Ian D. Chakeres and Charles E. Perkins. Dynamic MANET On-demand (DYMO) Routing. Internet draft, July 2007. draft-ietf-manet-dymo-10.txt.

[DD02] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Proc. FMCAD '02,* 4^{th} *Int. Conf. on Formal Methods in Computer-Aided Design*, pages 19–32. Springer, 2002.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS' 99* 14^{th} *IEEE Int. Symp. on Logic in Computer Science*, 1999.

[GBT] GBT - Graph Backwards Tool project home page. `http://www.it.uu.se/research/group/mobility/adhoc/gbt`.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[JK95] Bengt Jonsson and Lars Kempe. Verifying safety properties of a class of infinite-state distributed algorithms. In *Proc.* 7^{th} *Int. Conf. on Computer Aided Verification*, pages 42–53. Springer Verlag, 1995.

[JMB01] David B. Johnson, David A. Maltz, and Josh Broch. DSR: The dynamic source routing protocol for multi-hop wireless ad hoc

networks. In *Ad Hoc Networking*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[KK06] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. TACAS '06,* 12^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 197–211. Springer, 2006.

[KR06] H. Kastenberg and A. Rensink. Model checking dynamic states in GROOVE. In *SPIN Workshop*, pages 299–305. Springer, 2006.

[Lis] List of ad-hoc routing protocols - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Ad_hoc_routing_protocol_list`.

[PBR99] Charles E. Perkins and Elizabeth M. Belding-Royer. Ad-hoc on-demand distance vector routing. In *Proc.* 2^{nd} *Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pages 90–100. IEEE Computer Society, 1999.

[SWJ07] Mayank Saksena, Oskar Wibling, and Bengt Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. Technical Report 2007-035, Dept. of Information Technology, Uppsala University, Sweden, 2007.

[TGRW04] C. Tschudin, R. Gold, O. Rensfelt, and O. Wibling. LUNAR: a lightweight underlay network ad-hoc routing protocol and implementation. In *Proc. Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN)*, February 2004.

[The] The official IETF MANET working group web page. `http://www.ietf.org/html.charters/manet-charter.html`.

[ZP04] L. D. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3–4):139–169, 2004.