Liveness Checking as Safety Checking to Find Shortest Counterexamples to Linear Time Properties

Viktor Schuppan

Doctoral Thesis ETH No. 16268

Liveness Checking as Safety Checking to Find Shortest Counterexamples to Linear Time Properties

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Viktor Schuppan
Dipl.-Inf. Univ. TU München
born September 5, 1973
citizen of Germany

accepted on the recommendation of
Prof. Dr. Armin Biere, examiner
Prof. Dr. Daniel Kröning, co-examiner

2005

To those who seek the truth.

Abstract

Temporal logic is widely used for specifying hardware and software systems. Typically two types of properties are distinguished, safety and liveness properties. While safety can easily be checked by reachability analysis, and many efficient checkers for safety properties exist, more sophisticated algorithms have always been considered to be necessary for checking liveness. In this dissertation we describe an *efficient translation of liveness checking problems into safety checking problems* for finite state systems. More precisely, fair repeated reachability in a fair Kripke structure K is formulated as reachability in a transformed Kripke structure $K^{\mathbf{S}}$. A fair loop in K is detected in $K^{\mathbf{S}}$ by saving a previously visited state in an additional state-recording component, waiting until a fair state has been seen, and checking a loop closing condition. The approach extends to all ω-regular properties. We show that the size of the state space, the reachable state space, the transition relation, and its transitive closure grow by a factor of $|S|$ in the transformed model, where $|S|$ is the size of the state space in the original model. Radius and diameter increase by a small, constant factor. We discuss optimizations that limit the overhead of our translation. We have implemented the approach for BDD-based model checkers of the SMV family. Experimental results show not only that the approach is feasible for complex examples, but that it may lead to faster verification if the property turns out to be false. For one example even an exponential speed-up can be observed. We finally show that a similar reduction can be applied to a number of infinite state systems, namely, (ω-) regular model checking, pushdown systems, and timed automata.

Counterexamples as produced by a model checker for a failing property help developers to understand the problem in a faulty design. The shorter a counterexample, the easier it is typically to understand. The length of a counterexample, as reported by a model checker, depends on both the algorithm used for state space exploration and the way the property is encoded. We provide *necessary and sufficient criteria for a Büchi automaton to accept shortest counterexamples*. Extending a notion introduced by Kupferman and Vardi we call a Büchi automaton that accepts shortest counterexamples tight. We prove that Büchi automata constructed using the approach of Kesten et al. (KPR), which is essentially the same as the construction by Lichtenstein and Pnueli, are tight for future time LTL formulae, while an automaton generated with the algorithm of Gerth et al. (GPVW) may lead to unnecessary long counterexamples. Optimality is lost in the first case as soon as past time operators are included. We show that potential excess length is in both cases at most linear in the length of the specification. Using a recently proposed encoding for bounded model checking of LTL with past by Latvala et al., we construct a Büchi automaton that accepts shortest counterexamples for full LTL. The construction adapts the idea of virtual unrolling by Benedetti and Cimatti to Büchi automata. Its generalization gives a method to make an arbitrary Büchi automaton accept shortest counterexamples. We use our method of translating liveness into safety to find shortest counterexamples with a BDD-based symbolic model checker without modifying the model checker itself. Though

our method involves a quadratic blowup of the state space, it proves to be competitive with SAT-based bounded model checking. Experimental results show that using a model checking algorithm that finds shortest cycles contributes much more to a reduction in counterexample length than using an automaton that accepts shortest counterexamples when compared with the automaton by Kesten et al.

Zusammenfassung

Temporale Logik wird häufig für die Spezifikation von Hardware- und Softwaresystemen eingesetzt. Dabei wird oft zwischen Sicherheits- und Lebendigkeitseigenschaften unterschieden. Während Sicherheitseigenschaften einfach mittels Erreichbarkeitsanalyse überprüft werden können, wird Verifikation von Lebendigkeitseigenschaften meist mit spezialisierten Algorithmen in Verbindung gebracht. Dementsprechend sind mehr effiziente Werkzeuge zur Überprüfung von Sicherheitseigenschaften verfügbar als zur Überprüfung von Lebendigkeitseigenschaften. In der vorliegenden Dissertation wird eine Übersetzung entwickelt, die das Problem der Verifikation einer Lebendigkeitseigenschaft für Systeme mit endlich vielen Zuständen in das der Verifikation einer Sicherheitseigenschaft überführt. Genauer gesagt wird das Problem der Erreichbarkeit eines Zustandes s von sich selbst über einen fairen Pfad (faire wiederholte Erreichbarkeit) in einer fairen Kripke Struktur K in das Problem der (einfachen) Erreichbarkeit eines Zustandes $s^{\mathbf{S}}$ in einer anderen Kripke Struktur $K^{\mathbf{S}}$ übersetzt. Ein fairer Zyklus in K kann in $K^{\mathbf{S}}$ gefunden werden, indem zunächst nicht-deterministisch eine Kopie des momentanen Zustands s in einer separaten Version der Zustandsvariablen abgelegt wird. Sobald zunächst ein fairer und dann ein zu s identischer Zustand besucht worden sind, liegt ein fairer Zyklus vor. Die Übersetzung kann zur Überprüfung beliebiger ω-regulärer Eigenschaften verwendet werden. Die Größe des Zustandsraumes, des erreichbaren Zustandsraumes, der Übergangsrelation sowie ihrer transitiven Hülle im transformierten System wachsen proportional zur Anzahl Zustände im Originalsystem. Radius und Durchmesser ändern sich nur um einen kleinen, konstanten Faktor. Die Übersetzung wurde für Modellprüfer der SMV Familie implementiert. Es werden außerdem Optimierungen vorgestellt, die den Anstieg der Komplexität begrenzen helfen. Experimente zeigen, daß sich die transformierten Systeme nicht nur mit akzeptablem Aufwand verifizieren lassen, sondern daß sich Gegenbeispiele im transformierten System sogar manchmal schneller finden lassen als im Originalmodell. Ein Beispiel kann dabei sogar exponentiell schneller überprüft werden. Schließlich wird die Übersetzung auf einige Systeme mit unendlichem Zustandsraum erweitert, im einzelnen reguläre Modellprüfung, Kellerautomaten und Realzeitautomaten.

Die Gegenbeispiele, die ein Modellprüfer bei Fehlschlagen einer Spezifikation ausgibt, haben sich als sehr hilfreich bei der Fehlersuche erwiesen. Ein Gegenbeispiel ist umso einfacher zu verstehen, je kürzer es ist. Die Länge des ausgegebenen Gegenbeispiels hängt dabei sowohl vom Modellprüfungsalgorithmus als auch von der Repräsentation der Spezifikation als Büchi Automat ab. Es werden notwendige und hinreichende Kriterien entwickelt, die gewährleisten, daß ein Büchi Automat kürzeste Gegenbeispiele erkennt. Der Begriff der tightness von Kupferman und Vardi wird auf Büchi Automaten erweitert. Es wird gezeigt, daß ein Büchi Automat, der mit dem Algorithmus von Kesten et al. (KPR) konstruiert wird, kürzeste Gegenbeispiele für LTL eingeschränkt auf Zukunft erkennt. Im Gegensatz dazu führt der Algorithmus von Gerth et al. (GPVW) zu unnötig langen Gegenbeispielen. Die Optimalität bei KPR geht verloren, sobald

die Einschränkung auf Zukunft aufgehoben wird. Es konnte gezeigt werden, daß in beiden Fällen eine mögliche Überlänge höchstens linear in der Länge der Spezifikation ist. Durch Anpassung einer kürzlich vorgestellten Kodierung für Bounded Model Checking mit Vergangenheit von Latvala et al. wird ein Büchi Automat konstruiert, der kürzeste Gegenbeispiele für ganz LTL erkennt. Die Konstruktion überträgt die Idee des virtuellen Ausrollens von Benedetti und Cimatti auf Büchi Automaten. Ihre Verallgemeinerung ermöglicht es, einen beliebigen Büchi Automaten so zu verändern, daß er kürzeste Gegenbeispiele akzeptiert. Die oben beschriebene Übersetzung von Lebendigkeits- in Sicherheitseigenschaften kann nun benutzt werden, um kürzeste Gegenbeispiele mit einem BDD-basierten symbolischen Modellprüfer zu finden, ohne dabei den Programmkode des Modellprüfers selbst zu verändern. Trotz des quadratischen Wachstums im Zustandsraum werden Gegenbeispiele ähnlich schnell gefunden wie von einem SAT-basierten Pfadlängen-begrenzten Modellprüfer. Experimente zeigen, daß die Verkürzung in der Länge der Gegenbeispiele, die durch den Einsatz eines Modellprüfungsalgorithmus zum Finden kürzester Zyklen erreicht wird, weit größer ist als die, die aus dem Einsatz eines Büchi Automaten für kürzeste Gegenbeispiele resultiert.

Acknowledgments

First and foremost, I wish to thank my advisor Armin Biere. Being almost always available for discussion, he taught me his way of doing research and gave all required support. I doubt that there are many advisors who spend more time in direct contact with their students. I consider this an asset. Being able to attend one or two major conferences in the field per year, often without presenting a paper, also gave me very valuable input.

I would like to thank Daniel Kröning for being my co-advisor and providing support in spite of a tight schedule and pre-alpha drafts. Thomas Gross is to be thanked for taking over administrative responsibility during my final year.

My long-term office- and group-mate Cyrille Artho provided many thought-provoking impulses.

I also benefitted much from stimulating discussions with Keijo Heljanko, Tommi Junttila, Timo Latvala, and Andreas Podelski.

The software engineering group at ETH proved always open for visitors. Adam Darvas, Werner Dietl, and Bernd Schöller were very welcome links into that group.

Hans Dubach, Ruth Hidalgo, Eva Ruiz, and Hanni Sommer made my life at ETH easier by handling much of the administrative work.

The Computer Systems Institute was my home during my time at ETH. Seminars, coffee breaks, ski weekends, Assiabende, and, most important, lunch breaks — it was all there. Special thanks go to Hans Domjan, Philipp Kramer, Valery Naumov, Christoph von Praun, Patrik Reali, Florian Schneider, Yang Su, Cristian Tuduce, and Irina Tuduce for discussions, support, and friendship.

Last but not least, I am grateful to my parents and my friends for their patience and support.

Contents

Abstract **v**

Zusammenfassung **vii**

Acknowledgments **ix**

Contents **xi**

1 Introduction **1**
- 1.1 Safety and Liveness . . . 1
- 1.2 Counterexamples in Verification . . . 2
- 1.3 Thesis Statement and Contributions . . . 3
  - 1.3.1 Reduction . . . 3
  - 1.3.2 Büchi Automata for Shortest Counterexamples . . . 4
- 1.4 Outline . . . 5
- 1.5 Previously Published Results . . . 5

2 Common Concepts and Notation **7**
- 2.1 Background . . . 7
  - 2.1.1 Temporal Logic . . . 7
  - 2.1.2 Model Checking . . . 8
- 2.2 Preliminaries . . . 10
- 2.3 Kripke Structures as Models . . . 12
- 2.4 Linear Temporal Logic . . . 15
- 2.5 Büchi Automata . . . 17
- 2.6 Translating PLTLB Formulae into Büchi Automata . . . 18
- 2.7 Defining Safety and Liveness . . . 19
- 2.8 Model Checking Linear Time Properties . . . 21
  - 2.8.1 Basics . . . 21
  - 2.8.2 Lasso-shaped counterexamples . . . 22
  - 2.8.3 Model checking using BDDs . . . 22
  - 2.8.4 Bounded model checking using SAT solvers . . . 23
  - 2.8.5 Abstraction . . . 24

3 Symbolic Loop Detection for Finite State Systems **27**

3.1 Translating Simple Liveness into Safety 27
3.1.1 Intuition 27
3.1.2 Counter-Based Translation 30
3.1.3 State-Recording Translation 30
3.1.4 Comparison 31
3.2 Translating Fair Repeated Reachability 32
3.2.1 First Attempt 32
3.2.2 Optimization 32
3.2.3 Formalization and Correctness 34
3.2.4 Extensions 36
3.3 Complexity 36
3.3.1 Explicit State Model Checking 38
3.3.2 BDD-based Symbolic Model Checking 39
3.3.3 Summary 42
3.4 Shortest Counterexamples 43
3.5 Related Work 43
3.5.1 Reduction to and Power of Reachability Checking 43
3.5.2 Shortest Counterexamples 45
3.6 Summary 46

4 Extending to Infinite State Systems **47**
4.1 Regular Model Checking 47
4.1.1 Preliminaries 47
4.1.2 Reduction 48
4.1.3 Example 49
4.1.4 Discussion 51
4.2 Pushdown Systems 52
4.2.1 Preliminaries 52
4.2.2 Reduction 52
4.2.3 Complexity 55
4.2.4 Shortest Lasso-Shaped Counterexamples 58
4.3 Timed Automata 59
4.3.1 Preliminaries 59
4.3.2 Reduction 61
4.3.3 Complexity 65
4.3.4 Shortest Lasso-Shaped Counterexamples 65
4.4 Related Work 65
4.5 Summary 66

5 Büchi Automata for Shortest Counterexamples **67**
5.1 Tight Büchi Automata 67
5.2 (Non-) Optimality of Specific Approaches 71
5.2.1 Gerth et al. (GPVW) 71
5.2.2 Kesten et al. (KPR) 73

5.3 A Tight Look at LTL Model Checking . . . 74
5.3.1 Virtual Unrolling for Bounded Model Checking of PLTLB . . . 74
5.3.2 A Tight Büchi Automaton for PLTLB . . . 75
5.3.3 Partial Unrolling . . . 79
5.4 Generalization . . . 79
5.5 Related Work . . . 82
5.5.1 Virtual unrolling . . . 82
5.5.2 Tight automata . . . 83
5.5.3 Translating PLTLB into automata . . . 83
5.6 Summary . . . 84

6 Variable Optimization 85
6.1 The General Case . . . 85
6.2 Removing Constants . . . 86
6.3 Removing Input Variables . . . 87
6.4 Cone of Influence Reduction for Loop Detection . . . 88
6.5 Abstraction Refinement for Loop Detection . . . 90
6.6 Utility of 91
6.7 Related Work . . . 92
6.7.1 Completeness in bounded model checking . . . 92
6.7.2 Identifying input variables and variable dependencies . . . 94
6.7.3 Abstraction and refinement . . . 94
6.8 Summary . . . 96

7 Experiments 97
7.1 A Forward Jumping Counter . . . 97
7.2 Real-World Examples . . . 98
7.2.1 State-Recording Translation versus Standard Approach . . . 102
7.2.2 BDD- versus SAT-based Model Checking of the Tight Encoding . . . 102
7.2.3 The Cost of Tightness . . . 102
7.2.4 Comparing Variants of Variable Optimization . . . 105
7.2.5 A Tight Büchi Automaton in the Standard Approach . . . 105
7.3 Summary . . . 108

8 Conclusion 111
8.1 Contributions . . . 111
8.2 Future Work . . . 112
8.2.1 State-Recording Translation . . . 113
8.2.2 Infinite State Systems . . . 114
8.2.3 Tight Büchi Automata . . . 114

A Proofs and Auxiliary Lemmas 117

B Raw Data 119

List of Figures **127**

List of Tables **129**

Bibliography **131**

1

Introduction

While there's life, there's hope.
Cicero, Ad Atticum

1.1 Safety and Liveness

Informal characterization Two types of properties are frequently distinguished in temporal logic: *safety properties* state that "something bad does not happen," while *liveness properties* prescribe that "something good eventually happens" [Lam77]. Examples of safety properties are mutual exclusion, not exceeding a given resource bound, and partial correctness. Here, the "bad thing" is more than one process being in a critical section, using more resources than allowed, and terminating with a wrong result. "After the rain, the sunshine" [BBF+01], starvation freedom, and termination are examples of liveness properties. Clearly, the "good thing" is sunshine, a process making progress, and termination of a program. A *bounded liveness property* specifies that something good must happen within a given time, for example, "every request is followed by a reply within five units of time."*†

Safety is more useful, but ... Safety properties are considered more important in practice than liveness properties. They are most crucial to system correctness and, therefore, deserve higher priority and more time in verification [BBF+01]. In fact, "More than 90% ... of the errors in real systems are violations of safety properties" [Lam04]. However, inferior performance of checking liveness properties also contributes to the fact that some engineers do not bother to specify liveness properties [BL03, BBF+01, Lar04]: the models on which safety properties are checked may be too large to verify liveness properties; as a consequence, models used to check liveness properties may lack sufficient level of detail to find subtle errors [Lam04]. Nevertheless, as every safety property is satisfied by the empty system [SL88], some form of liveness is clearly desirable. Or, in Neil Jones' words [Jon04]:

> Without safety, liveness is illusory;
> Without liveness, safety is ephemeral.

This leaves the choice of either bounded or unbounded liveness.

*For clarification we sometimes refer to liveness properties as unbounded liveness properties.

†Technically, bounded liveness properties are safety properties (see Sect. 2.7). In particular, algorithms for checking safety can be used to check bounded liveness. In this chapter we prefer to treat them as a separate type of properties, which are amenable to safety checking but are liveness properties in their purpose.

Bounded liveness versus unbounded liveness In real life, unbounded liveness — only knowing that something good will happen eventually — is useless [BBF+01, PSZ]. At some stages during development, unbounded liveness may nevertheless be the preferred level of abstraction [BBF+01]. First, unbounded liveness is easier to read and write [BBF+01, Var04]. This is especially true if parameters are to be included in bounded specifications of parameterized systems [BBF+01]. Second, timed behavior means high complexity [BBF+01, GGA05]. Third, when working with concurrent and distributed systems, the notion of a next state may not be meaningful [Lam83, PSZ]. Once unbounded liveness has been established, other means, such as testing, may be used to check for bottlenecks if average performance is the primary concern [Lam04]. Finally, bounded and unbounded liveness need not be contradictory [BBF+01]: truth of bounded liveness implies truth of unbounded liveness; conversely, a proof of unbounded liveness may be used to extract bounds.

Why bother? Lamport's original motivation to distinguish safety and liveness properties was that they typically require different proof techniques [Lam77]. In deductive verification, safety properties are usually proved using invariance arguments, while liveness properties are verified with a well-foundedness argument [OL82]. In model checking, reachability is sufficient to check safety properties [KV01], while liveness properties require some form of cycle detection [VW86, EL87, BCCZ99]. Furthermore, safety properties are deemed more important and easier to verify; hence, more effort should be spent on them [BBF+01]. Classifying properties can also help to come up with better structured specifications and may lead to fewer omissions [BBF+01]. Fairness constraints need to be taken into account only when proving liveness properties [Sis94]. The modeling stage may be affected by the kind of property as well: different simplifications preserve different classes of properties [BBF+01]. Finally, it is easy to monitor executions for violations of safety properties [Sis94, HR02].

Is this the right distinction? Manna and Pnueli remark that the important distinction may be between safety and non-safety [MP90]. However, Alpern and Schneider show that every ω-regular property — the class of properties that we are concerned with — is the intersection of a safety and a liveness property [AS87]. Hence, liveness seems to capture in a most general sense the idea of what may be the non-safety part of an ω-regular property. Purpose of the above discussion was to establish that it is useful to look beyond safety.

Problem statement Safety and more general ω-regular properties are treated differently in a number of proofs, algorithms, and tools. Often, a technique or a tool is presented first for safety properties and is only later extended to handle more properties. Examples are symbolic trajectory evaluation [SB95, YS01], regular model checking [KMM+01, BJNT00, PS00], and UPPAAL [LPY97, BDL04]. Some optimizations, such as forward model checking [INH96, HKQ98, BCZ99], are only available when checking safety properties.

1.2 Counterexamples in Verification

Usefulness of counterexamples In the last years there has been an increased adoption of formal methods, especially model checking [CGP99], by the hard- and software industry [BDEGW03, Ben01, BR02]. Automation, limitation of scope to high-risk components, and a focus on bug finding rather than full verification have contributed to that [BCLR04, Sch03, AVARB+01]. Counterexamples [CV03] show a (partial) execution that exhibits an error. They are provided by model checkers when the verification of a property fails and have proved helpful in both, finding the actual fault and improving verification methodology. The utility of counterexamples is witnessed by the success of SAT-based bounded model checking [BCCZ99]: initially, bounded model checking could only be used to find bugs; nevertheless, the technology made a very quick transition into industry [CFF+01]. Some researchers even claim that counterexamples are the "single most effective feature to convince system engineers about the value of formal verification" [CV03]. With respect to methodology, recent progress in model checking software is largely based on an automated abstraction-refinement cycle, which uses counterexamples to guide the refinement [CGJ+03].

Problem statement Most counterexamples still need to be interpreted by humans. Shorter counterexamples will, in general, be easier to understand. In the automata-theoretic approach to model checking [VW86], finding a shortest counterexample to an ω-regular property requires both, a method to find shortest cycles and a suitable Büchi automaton to encode the property. While SAT-based bounded model checkers can find shortest counterexamples, SAT-based bounded model checking does not perform equally well on all examples as BDD-based symbolic model checking and vice versa [AS04]. An efficient BDD-based technique that produces shortest counterexamples has not been available so far. No suitable Büchi automaton to encode the property has been extracted from the custom encodings used by SAT-based bounded model checkers.

1.3 Thesis Statement and Contributions

In this dissertation we establish the following thesis:

1. For finite state systems, verification of ω-regular properties using repeated reachability can be syntactically reduced to verification of safety properties using reachability. The reduction leads to a quadratic increase in system size. It extends to a number of infinite state systems.

2. Necessary and sufficient criteria for Büchi automata to accept shortest counterexamples to LTL with past can be stated. A Büchi automaton fulfilling these criteria can be constructed. Its combination with the reduction from repeated reachability to reachability gives a practical method to find shortest counterexamples to LTL with past.

1.3.1 Reduction

Basic idea We develop a reduction from fair repeated reachability to reachability. It takes a finite state system M equipped with a set of fairness constraints F and produces another finite state system M' such that there is an initialized fair path in M iff a certain set of states in M' is reachable. The basic idea is borrowed from explicit on-the-fly model checking [CVWY92] and from bounded model checking [BCCZ99]: a counterexample to a simple liveness property $\mathbf{F}p$ ("finally p") in a finite state system is lasso-shaped, i.e., it consists of a stem that leads to a loop such that p is false on stem and loop. As in [BCCZ99] the major challenge is how to detect the loop. Our translation tries to guess the start of a loop, saves it in a copy of the state variables, and checks whether the saved state occurs a second time. When this happens, a loop has been found and the property is checked. Via a standard automaton construction [VW94] our translation is applicable to all ω-regular properties.

Source-to-source Our reduction is source-to-source and can be applied even manually on the design entry-level. The user does not need to have access to the source code of the tool itself. This could be useful in an industrial setting, where the source code of a tool is usually not available. To some extent it might also discourage tool vendors to charge extra license fees for liveness support, if compromises with respect to capacity are acceptable.

Complexity Saving of the state variables doubles the number of state variables in the reduced system. Disregarding a (small constant) number of additional flags, the number of states of the reduced system is the square of the number of states in the original system. We also establish bounds on the number of forward iterations required to check the reduced system.

Forward model checking, shortest cycles The reduced system can be verified using forward model checking [INH96, HKQ98, BCZ99]. When this is applied, shortest cycles are obtained in finite state systems.

Performance Experiments with NuSMV [CCG+02] and Cadence SMV [McM] show that a reachability algorithm can check the reduced system with acceptable overhead compared to the traditional algorithm on the original system. This is in part due to specific optimizations that target the overhead introduced by the reduction. In some cases the optimizations give improvements of more than 2 orders of magnitude over the unoptimized reduction. On specific examples our approach is exponentially faster than the traditional algorithm.

Infinite state systems The reduction extends to ($\omega-$)regular model checking [KMM+01, WB98, BJNT00, BLW04a], pushdown systems [BEM97, FWW97, EHRS00a], and timed automata [AD94].

1.3.2 Büchi Automata for Shortest Counterexamples

Tight Büchi automata We extend the notion of Kupferman and Vardi of a tight automaton on finite words [KV01], which accepts shortest violating prefixes for safety properties, to Büchi automata. A tight Büchi automaton accepts shortest counterexamples to ω-regular properties. We establish necessary and sufficient criteria for a Büchi automaton to be tight.

Results for known automata constructions A simple example shows that the translation from LTL to Büchi automata by Gerth et al. [GPVW96] and some of its descendants do not generate tight Büchi automata. Resulting counterexamples can have excess length linear in the length of the formula. The translation by Kesten et al. [KPR98] produces tight Büchi automata for future time LTL but exhibits excess length linear in the number of past time operators if those are admitted.

Constructing tight Büchi automata We show how to construct a tight Büchi automaton for LTL formulae with past. The construction is based on an encoding of LTL with past for bounded model checking [LBHJ05]. We generalize the construction to make arbitrary Büchi automata tight.

Performance Experimental results show that finding shortest counterexamples with a BDD-based symbolic model checker using the reduction from repeated reachability to reachability and a tight Büchi automaton based on [LBHJ05] is competitive with SAT-based bounded model checking.

1.4 Outline

The outline of this dissertation is as follows. The next chapter 2 discusses some required background and introduces notation. Chapter 3 presents the reduction from fair repeated reachability to reachability. It is extended to some infinite state systems in Chap. 4. Büchi automata for shortest counterexamples are discussed in Chap. 5. In Chap. 6 variable optimization is proposed to alleviate the overhead introduced by the reduction. Chapter 7 presents experimental results that show the viability of our approach. Chapter 8 concludes.

1.5 Previously Published Results

This dissertation is partially based on the following publications. The initial reduction of repeated reachability to reachability is presented in [BAS02]. A follow-up journal article [SB04] adds optimizations and a more extensive experimental evaluation. It also gives a corrected and more detailed analysis of the complexity of the reduction. The extension to infinite state systems is contained in [SB06]. The investigation of Büchi automata for shortest counterexamples appeared in [SB05]. Some of the examples are taken from a case study [SB03].

2

Common Concepts and Notation

In the history of mankind, no two people have ever been able to agree on the toppings for pizza.

Jim Davis, Garfield

To verify a software or hardware system using model checking the system is *modeled* in a form understood by the model checker, a *specification* is given as a set of *properties*, and finally a model checker is employed to *verify* that the specification holds in the model or to provide a counterexample, which shows why the specification is wrong. After giving some background and preliminaries we discuss concepts and notation used in each task in turn. Localized notation is presented in subsequent chapters as needed.

2.1 Background

2.1.1 Temporal Logic

Motivation A broad class of programs is *transformational* [HP85] in character: they take inputs, compute functions on the inputs, and output the results. Specifying and reasoning about transformational programs, which are usually sequential and terminating, can be done, e.g., with the help of pre- and postconditions and Hoare triples [Flo67, Hoa69]. However, these may not be sufficient for *reactive* programs [HP85]: these are designed to constantly interact with their environment, such as operating systems, and are often concurrent or designed not to terminate. For such programs one also wishes to specify and reason about what can happen in intermediate states of a — potentially infinite — execution. Pnueli has established *temporal logics* for specifying properties of and reasoning about such programs [Pnu77]. Temporal logics are a special kind of modal logics that include operators ("modalities") to reason about the truth values of assertions at different times during the execution of a program. A survey on temporal logic is available in [Eme90].

Linear versus branching time Temporal logics distinguish a *linear* and a *branching* view on time [Lam80]. In the linear view, each point in time has exactly one future. A specification is interpreted over a linear structure, i.e., a computation is a sequence of events. *LTL* [Pnu77, MP83, Eme90] is an example of a linear temporal logic. In the branching view, there is a (non-deterministic) choice between several potential futures at each point in time. This results in a tree of potential computations. A temporal logic providing a pure branching view is *CTL* [CE82, Eme90]. Neither view can, on its own, express all properties that the other can [Lam80, EH86]. CTL^* [EH86] and the *μ-calculus* [Koz83] integrate both views. However, few tools actually support these [Bie97]. While model checking linear time has complexity linear in the size of the model and exponential in the size of the formula [LP85], model checking branching time is linear in both the size of the model and the size of the formula [CES86]. In practice, size and structure of the model are at least equally important and worst case behavior is rare [Hol03, CGH97]. Users of model checking tools tend to prefer specifications in linear time temporal logic [Var01]. For more discussion and arguments in favor of the linear view and more references see [Var01, Hol03]. The main reason for focusing on the linear view below is simply the fact that repeated reachability is the main vehicle used in model checking linear time and, hence, the proposed techniques are readily applicable there.

Usefulness of past operators Temporal logics as originally developed by philosophers included both *past* and *future operators* [Eme90]. It turns out that past operators do not add expressive power when reasoning about presence and future only, which is the case when specifying properties of initialized computation paths [Kam68, GPSS80]. Consequentially, past operators were initially not included in temporal logics for verification (e.g. [Pnu77]). However, as argued by [LPZ85], past operators can replace history variables [OG76] in modular verification. Further, some specifications are more natural to write if past operators are available [LPZ85] — in fact, some specifications can be made exponentially more succinct [LMS02]. Finally, past operators do not increase worst-case complexity of the model checking problem (see Sect. 2.8) [SC85]. Hence, mostly stemming from [LP85], extensions to past have been described and implemented for explicit state [GO03], symbolic SAT-based [BC03, CRS04, LBHJ05]*, and symbolic BDD-based model checking [FMPT01]. NuSMV [CCG+02] contains some of these implementations.

2.1.2 Model Checking

Motivation Traditional deductive reasoning (e.g., [Pel01]) about temporal properties of hardware and software systems is not well automated and, therefore, considered difficult and tedious. If a system is — or can be abstracted to be — finite state, reasoning can be automated: the system can be represented as a finite state graph and graph-theoretic algorithms can be employed to determine truth of temporal properties. This approach, termed *model checking*, was pioneered independently by Clarke and Emerson [CE82] and Queille and Sifakis [QS82]. Automation doesn't come for free, though: the size of the state graph can be exponential in the description of the system (called the "state explosion problem"), and infinite state systems cannot be handled without further measures. Consequently, a significant amount of research in model checking has been devoted to both problems. In spite of these challenges, model checking has been widely adopted in the hardware and software industry, e.g., at IBM [AVARB+01, BDEGW03], Intel [Ben01, Sch03], and Microsoft [BR02, BCLR04]. Or, as Jackson and Rinard put it [JR00]:

> Indeed, the success of model checking ... can largely be credited with saving the reputation of formal methods.

*Note that [CRS04] contains a bug and may produce incorrect results when the property uses past operators [LBHJ05]. For a fix see [BHJ+06].

A survey on model checking is [CS01], for textbooks see [CGP99, BBF+01, Hol03, Pel01].

Counterexamples A particularly useful feature of model checkers is the generation of error traces or *counterexamples*, see Sect. 1.2.

Automata-theoretic approach Many algorithms for model checking temporal formulae represent the system as a graph and work through the formula syntactically (e.g., [LP85, CES86]). Vardi and Wolper came up with a different approach [VW86]: if the model is regarded as a language generator and the property as a language acceptor, one can represent both, model and property, as automata on infinite objects where the type of the automata depends on the system and on the logic at hand. Assuming that the model is already represented as an automaton of the appropriate kind, this makes model checking a two-step process: first, translate the formula into a corresponding automaton; second, solve an automata-theoretic question, which involves both automata, such as language emptiness of the intersection of both automata. This approach makes many results from automata theory immediately available to model checking and often leads to asymptotically optimal algorithms [KVW00]. The automata-theoretic approach is dominating the verification of linear temporal logic, while the syntactic approach is preferred for branching time.

Explicit state model checking The first model checkers were *explicit state model checkers*, i.e., they used an explicit representation of both transition relation and sets of states. This implies that each pair of states in the transition relation and each state in a set of states take up a non-zero amount of memory. The state explosion problem severely limits the size of systems that can be handled with this approach in its pure form.

BDD-based symbolic model checking *Symbolic model checkers* represent the transition relation and sets of states symbolically and operate on these symbolic representations. In a system with a high degree of regularity this can yield exponential savings in memory. Many researchers used *binary decision diagrams* (BDDs) [Bry86] in verification, for examples see [BCM+92, BF90, Pix91, CMB91]. Among those, the work of McMillan [McM93], leading to the SMV model checker, proved most successful.

SAT-based symbolic model checking The increasing power of Boolean SAT-solvers spurred the development of *SAT-based symbolic model checkers* using propositional Boolean formulae as representation of state transition systems [BCCZ99, BCC+99, BCRZ99]. Typically, SAT-based model checkers are *bounded*, i.e., they check paths only up to a user-defined length. This makes them most suitable for bug finding. Techniques to overcome this limitation are an active area of research [SSS00, McM03, AS04, HJL05]. For a recent survey on SAT-based formal verification see [PBG05].

Abstraction Many authors consider *abstraction* as one of the most powerful tools to combat the state explosion problem [CGP99, Hol03]. Abstraction (see, e.g., [CGL94, Kur94, CC77, Kro99]) can be used to abstract from unnecessary details of the state space, leading to a reduction in the number of states. *Predicate abstraction* is especially useful to obtain finite approximations of infinite state systems [GS97]. A too coarse abstraction may not allow to prove or disprove a property, while a too fine abstraction can make verification intractable. Automated *abstraction refinement* tries to alleviate this problem by starting with a coarse abstraction and subsequently refining it based on information from unsuccessful verification attempts. Examples of approaches where refinement is driven by abstract counterexamples that have no counterpart in the concrete model (called *spurious counterexamples*) are the work by Balarin and Sangiovanni-Vincentelli [BSV93], Kurshan's automatic localization reduction ([Kur94], pp. 170–172), and counterexample guided abstraction refinement by Clarke et al. [CGJ+03].

Further approaches to state explosion *Partial order methods* are another technique to combat state explosion [GP93, Pel96, Val92]. They exploit concurrency among asynchronously executing parts of a system by taking only one order of independent transitions into account if other orders of these transitions are known to lead to the same state. Other successful approaches to the state explosion problem include *modular reasoning*, *symmetry reduction*, and *induction*. For references see [CGP99, CS01].

Application areas, examples Partial order methods are particularly successful in the verification of asynchronous concurrent systems, which are typically software. Examples of explicit state model checkers employing partial order reduction are SPIN [Hol03], Bogor [RDH03], and Java PathFinder [VHB+03]. Symbolic model checkers are most suitable in a synchronous — typically hardware — setting. The original SMV [McM93], its popular open-source reimplementation NuSMV [CCG+02], an industrial successor at Cadence [McM], or VIS [VIS96] are BDD-based symbolic model checkers belonging to this class. NuSMV also includes SAT-based bounded and unbounded model checking routines [CRS04, HJL05]. CBMC is a SAT-based symbolic bounded model checker for ANSI-C [CKL04]. Based on predicate abstraction and counterexample guided abstraction refinement, symbolic methods have made symbolic model checking efficient for software as well. Examples are SLAM [BR02], BLAST [HJMS02], and MAGIC [CCG+04].

2.2 Preliminaries

Basics The set of *Booleans* is denoted by $\mathbb{B} = \{0, 1\}$; $\mathbb{N}$ and $\mathbb{R}$ are *naturals* and *reals*, respectively. Tuples are enclosed in parentheses, elements of a tuple are separated by commas.

Sequence Let Σ be a finite alphabet, let α be a finite or infinite sequence over Σ. Elements of a sequence typically have no operator between them, as have subsequences that are *concatenated* to form a larger sequence. If ambiguity might arise, $\circ$ is used to denote concatenation of elements and/or (sub)sequences. Concatenation is defined if the left operand has finite length. The *length* of a sequence α is defined as its number of elements, i.e., $|\alpha| = n + 1$ if $\alpha = \sigma_0\sigma_1 \ldots \sigma_n$ is finite, ∞ otherwise. $\alpha[i]$ denotes the element at index i where counting starts from 0. $\alpha[i, j]$ is the subsequence $\alpha[i]\alpha[i + 1] \ldots \alpha[j]$ of α, where $\alpha[i, j] = \epsilon$ if $i > j$. $\alpha[i, \infty]$ is α with its first i states chopped off. α is a *prefix* of β, denoted $\alpha \sqsubseteq \beta$, iff $\alpha = \epsilon \vee \exists 0 \leq i < |\beta| \;.\; \alpha = \beta[0, i]$. If S is a set of sequences, the set of *finite prefixes* of (members of) S is $pre(S) = \{\alpha \mid |\alpha| < \infty \wedge \exists \beta \in S \;.\; \alpha \sqsubseteq \beta\}$. $inf(\alpha)$ denotes the set of elements appearing infinitely often in α. The cross product of two sequences α, β of equal length, $\alpha \times \beta$, is defined component-wise.

Composing (sets of) sequences Let S be a set of finite sequences and T be a set of finite or infinite sequences, both over Σ. If S or T are sets of elements of Σ, consider them to be sets of sequences of length 1. Then $S \circ T = \{s \circ t \mid s \in S \wedge t \in T\}$ denotes the set of sequences concatenated from elements of S and T. Let α be a finite sequence over Σ. Then α^* (resp. α^+) denotes the set of sequences obtained by finite (resp. finite non-zero) repetition of α. If $\alpha \neq \epsilon$, α^ω is the sequence obtained by infinite repetition of α. The operators $^*, ^+, ^\omega$ are extended to sets of finite (non-empty) sequences in the natural way.

Language A language over Σ is a subset of $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. An ω-language is a subset of Σ^ω.

Regular expressions/languages A *regular expression* over a finite alphabet Σ is defined inductively as follows:

1. $\emptyset$ is a regular expression (*empty set*),
2. $\{\epsilon\}$ is a regular expression (*empty sequence*),
3. $\forall a \in \Sigma$, a is a regular expression (*singleton*),
4. if S and T are regular expressions, $S \circ T$ is a regular expression (*concatenation*),
5. if S and T are regular expressions, $S \cup T$ is a regular expression (*union*, *alternative*), and
6. if S is a regular expression, S^* is a regular expression (finite but unbounded repetition, *Kleene star*).

In addition, brackets may be used to indicate operator precedence. Each regular expression induces a language $R \subseteq \Sigma^*$, called a *regular language*. The set of regular languages over Σ is closed under intersection and complement. We identify a regular expression with the language it induces.

ω-regular expressions/languages *ω-regular* expressions extend regular expressions to obtain languages of infinite words, called *ω-regular languages*:

1. if S is a regular expression, S^ω is an ω-regular expression,
2. if S is a regular expression and T is an ω-regular expression, $S \circ T$ is an ω-regular expression, and
3. if S and T are ω-regular expressions, $S \cup T$ is an ω-regular expression.

ω-regular languages are also closed under intersection and complement.

Lassos Let β, γ with $\gamma \neq \epsilon$ be finite sequences. A sequence α is a $\langle\beta,\gamma\rangle$-*lasso* with *stem* β and *loop* γ iff $\alpha = \beta\gamma^\omega$. We sometimes write $\langle\beta,\gamma\rangle$ instead of $\beta\gamma^\omega$. A sequence α is *lasso-shaped* iff there exist β,γ such that α is a $\langle\beta,\gamma\rangle$-lasso. The *length* of a lasso is defined as $|\langle\beta,\gamma\rangle| = |\beta| + |\gamma|$. A lasso $\langle\beta,\gamma\rangle$ is *minimal* for α iff $\alpha = \beta\gamma^\omega$ and $\forall\beta',\gamma' \,.\, \alpha = \beta'\gamma'^\omega \Rightarrow |\langle\beta,\gamma\rangle| \leq |\langle\beta',\gamma'\rangle|$. The *type* [LMS02] of a $\langle\beta,\gamma\rangle$-lasso is defined as $\mathit{type}(\langle\beta,\gamma\rangle) = (|\beta|,|\gamma|)$. A sequence α can be mapped to a set of types: $\mathit{type}(\alpha) = \{\mathit{type}(\langle\beta,\gamma\rangle) \mid \alpha = \beta\gamma^\omega\}$. We state the following facts about sequences (proved in appendix A).

Lemma 1 *Let $\langle\beta,\gamma\rangle$ be a minimal lasso for α, $\langle\beta',\gamma'\rangle$ a minimal lasso for α', and $\alpha'' = \alpha \times \alpha'$. Then there are finite sequences β'',γ'' such that $\langle\beta'',\gamma''\rangle$ is a minimal lasso for α'', $|\beta''| = \max(|\beta|,|\beta'|)$, and $|\gamma''| = \mathrm{lcm}(|\gamma|,|\gamma'|)$.*[†]

Lemma 2 *Let $\alpha = \beta\gamma^\omega = \beta'\gamma'^\omega$ with $\langle\beta,\gamma\rangle$ minimal for α. Then $|\beta| \leq |\beta'|$ and $|\gamma|$ divides $|\gamma'|$.*

Corollary 3 *Let $\langle\beta,\gamma\rangle$ be minimal for α. $\langle\beta,\gamma\rangle$ is unique.*

2.3 Kripke Structures as Models

Kripke structure Literature on model checking uses state-labeled transition systems as models, called Kripke structures [MOSS99, CGP99]. Let AP be a set of *atomic propositions*. A *Kripke structure* over AP is a four tuple $K = (S,T,I,L)$ where S is a finite set of *states*, $T \subseteq S \times S$ is a *transition relation*, $I \subseteq S$ is the set of *initial states*, and $L \mapsto 2^{AP}$ is a *labeling* of the states with the subset of atomic propositions true in that state. A *fair* Kripke structure has an additional finite set of (weak) *fairness constraints* $F = \{F_0,\ldots,F_f\}$ with $F_i \subseteq S$ for $0 \leq i \leq f$. A primed state s' denotes a *successor state* or *next state* of some state s, i.e. $(s,s') \in T$. T^+ denotes the transitive, T^* the reflexive, transitive closure of the transition relation.

Symbolic notation It is often more convenient (e.g., [KPR98]) to construct the state space of a Kripke structure as the set of valuations of a set of *state variables* V, with each $v_i \in V$ ranging over a set of values V_i. If s and s' are current and next states, $v_i(s)$ and $v_i(s')$ denote the valuation of v_i in s and s', respectively. When s and s' are clear from the context, we often write v_i and v_i'. v_i and v_i' are also called *current-state variable* and *next-state variable*. When we use the symbolic notation, we write $K = (V,S,T,I,L,F)$ where V is the set of state variables, S is a predicate that restricts the potential valuations of the variables in V (called a *state invariant*), and T, I, L, and F retain their usual meaning, but are typically given as predicates over current- and next-state variables.

Path A non-empty sequence of states π is a *path* in K if $\forall 0 < i < |\pi| \,.\, (\pi[i-1],\pi[i]) \in T$.[‡] If $\pi[0] \in I$, π is *initialized*. An infinite path is *fair* if $\forall F_i \in F \,.\, \forall j \geq 0 \,.\, \exists k > j \,.\, \pi[k] \in F_i$. Π

[†] $\mathrm{lcm}(a,b)$ denotes the *least common multiple* of a and b.

[‡] Contrary to some other work in model checking, our definition of path length counts states rather than transitions. While counting transitions is closer to the notion of distance, that is most relevant when weighted transitions are used. As we do not need weighted transitions and counting states simplifies some of the technical parts (e.g., the length of two concatenated sequences of states is simply the sum of their lengths) we prefer to count states rather than transitions.

denotes the set of all initialized paths in K, $\Pi^F \subseteq \Pi$ the set of initialized fair paths. Note, that if $F = \emptyset$ every infinite path is considered fair.

Reachability Let $s_1, s_2 \in S$ be states and $S_1 \subseteq S$ a non-empty set of states. s_2 is *reachable* from s_1 iff there exists a finite path π in K that starts in s_1 and ends in s_2: $\exists \pi \,.\, |\pi| < \infty \wedge \pi[0] = s_1 \wedge \pi[|\pi|-1] = s_2$. s_2 is reachable from a set of states S_1 iff it is reachable from some state in S_1. Reachability of a set of states from an individual state or a set of states is defined correspondingly. The set of reachable states of K, $R(K)$, is defined as $R(K) = \{s \in S \mid s \text{ is reachable from } I\}$.

Strongly connected component A non-empty set of states $S_1 \subseteq S$ is a *strongly connected component* (SCC) iff for each pair of states $s_1, s_2 \in S_1$, s_2 is reachable from s_1. An SCC is *non-trivial* iff it consists of more than one state or of a single state with a transition to itself. A strongly connected component S_1 is *fair* iff it intersects each fairness constraint: $\forall 0 \leq i \leq f \,.\, S_1 \cap F_i \neq \emptyset$.

Distance, radius, diameter The *distance* of a state t from a state s in K, written $\delta(K,s,t)$, is the length of a shortest path in K that starts in s and ends in t: $\delta(K,s,t) = \min\{|\pi| \mid \pi[0] = s \wedge \pi[|\pi|-1] = t\}$. $\delta(K,s,t)$ is defined to be ∞ if t is not reachable from s. The distance of a state t from a non-empty set of states S_1 is the minimal distance of t from all states $s \in S_1$. The *radius* of a Kripke structure is the maximum distance of any state from the initial states: $r(K) = \max\{\delta(K,I,s) \mid s \in R(K)\}$. The *diameter* of a Kripke structure K is the length of a longest loop-free path in K: $d(K) = \max\{|\pi| \mid \forall 0 \leq i < j < |\pi| \,.\, \pi[i], \pi[j] \in R(K) \wedge \pi[i] \neq \pi[j]\}$.

Projection Sometimes we only want to take a subset of state variables into account. Let s be a state over a set of state variables V. Let $\tilde{V} = \{v_{i_0}, \ldots, v_{i_{|\tilde{V}|-1}}\} \subseteq V$ be a subset of state variables. The *projection* of s onto $\tilde{V}$ is defined as $s|_{\tilde{V}} = (v_{i_0}(s), \ldots, v_{i_{|\tilde{V}|-1}}(s))$. The definition is extended to paths and sets of states in the natural way.

Language Each fair path implicitly defines an infinite sequence over 2^{AP}. We write $L(\pi)$ for the sequence over 2^{AP} induced by π: $\alpha = L(\pi) \Leftrightarrow \forall 0 \leq i \,.\, \alpha[i] = L(\pi[i])$. The *language* of a Kripke structure K over AP is defined as $\mathit{Lang}(K) = \{L(\pi) \mid \pi \in \Pi^F\}$. Sometimes it is useful to consider the language of all fair paths starting from a particular state s: for $s \in S$, we define $\mathit{Lang}(K,s) = \mathit{Lang}((S,T,\{s\},L,F))$.

Property, satisfaction Similarly, a *property* ϕ is an ω-language over 2^{AP}.[§] A property ϕ *holds universally* in a Kripke structure K (with both ϕ and K over AP), denoted $K \models_\forall \phi$, iff $\mathit{Lang}(K) \subseteq \phi$. It *holds existentially*, written $K \models_\exists \phi$, iff $\mathit{Lang}(K) \cap \phi \neq \emptyset$. If $K \not\models_\forall \phi$ then each path $\pi \in \Pi^F$ with $L(\pi) \notin \phi$ is a *counterexample* to the property. A *witness* is defined analogously in the existential case. In all subsequent chapters we are only concerned with the universal case and write $\models$ when we mean $\models_\forall$. A property is ω*-regular* if it is an ω-regular language.

[§] See Sect. 2.1.1 for a discussion on that choice.

Sum The *sum* K_3 of Kripke structures K_1 and K_2 can exhibit the behavior of either of its constituents, i.e., it is defined with language union in mind. If the sets of atomic propositions AP_1 and AP_2 of K_1 and K_2 don't match, any word in $Lang(K_3)$ behaves like a word in the language of K_i w.r.t. AP_i and is free w.r.t. $AP_{3-i} \setminus AP_i$ for $i \in \{1,2\}$. Formally, let $K_1 = (S_1, T_1, I_1, L_1, F_1 = \{F_{1,0}, \ldots, F_{1,f_1}\})$ and $K_2 = (S_2, T_2, I_2, L_2, F_2 = \{F_{2,0}, \ldots, F_{2,f_2}\})$ be Kripke structures over AP_1 and AP_2. The sum $K_3 = K_1 + K_2$ is defined as $K_3 = (S_3, T_3, I_3, L_3, F_3)$ over $AP_3 = AP_1 \cup AP_2$ where

$$
\begin{array}{lcl}
S_3 & = & S_1 \times 2^{AP_2 \setminus AP_1} \cup 2^{AP_1 \setminus AP_2} \times S_2 \\
T_3 & = & \{((s_1, s), (s_1', s')) \in S_3 \times S_3 \mid (s_1, s_1') \in T_1\} \cup \\
 & & \{((s, s_2), (s', s_2')) \in S_3 \times S_3 \mid (s_2, s_2') \in T_2\} \\
I_3 & = & I_1 \times 2^{AP_2 \setminus AP_1} \cup 2^{AP_1 \setminus AP_2} \times I_2 \\
L_3((s_1, s_2)) & = & L_1(s_1) \cup s \quad \text{if} \quad (s_1, s_2) \in S_1 \times 2^{AP_2 \setminus AP_1}, \\
 & & s \cup L_2(s_2) \quad \text{otherwise, and,} \\
F_3 & = & \{\tilde{F}_{1,0}, \ldots, \tilde{F}_{1,f_1}, \tilde{F}_{2,0}, \ldots, \tilde{F}_{2,f_2}\} \quad \text{with} \\
 & & \quad \tilde{F}_{1,j} = F_{1,j} \times 2^{AP_2 \setminus AP_1} \cup 2^{AP_1 \setminus AP_2} \times S_2, \\
 & & \quad \tilde{F}_{2,j} = S_1 \times 2^{AP_2 \setminus AP_1} \cup 2^{AP_1 \setminus AP_2} \times \in F_{2,j}
\end{array}
$$

Synchronous product The *synchronous product* of Kripke structures is defined over tuples of states, where shared atomic propositions of the component states in a tuple have to match. Hence, the idea is language intersection. Formally, let $K_1 = (S_1, T_1, I_1, L_1, F_1 = \{F_{1,0}, \ldots, F_{1,f_1}\})$ and $K_2 = (S_2, T_2, I_2, L_2, F_2 = \{F_{2,0}, \ldots, F_{2,f_2}\})$ be Kripke structures over AP_1 and AP_2. The synchronous product of K_1 and K_2 is defined as $K_3 = (S_3, T_3, I_3, L_3, F_3)$ over $AP_3 = AP_1 \cup AP_2$ where

$$
\begin{array}{lcl}
S_3 & = & \{(s_1, s_2) \in S_1 \times S_2 \mid L(s_1) \cap AP_2 = L(s_2) \cap AP_1\}, \\
T_3 & = & \{((s_1, s_2), (s_1', s_2')) \in S_3 \times S_3 \mid (s_1, s_1') \in T_1 \wedge (s_2, s_2') \in T_2\}, \\
I_3 & = & \{(s_1, s_2) \in S_3 \mid s_1 \in I_1 \wedge s_2 \in I_2\}, \\
L_3((s_1, s_2)) & = & L_1(s_1) \cup L_2(s_2), \quad \text{and} \\
F_3 & = & \{\tilde{F}_{1,0}, \ldots, \tilde{F}_{1,f_1}, \tilde{F}_{2,0}, \ldots, \tilde{F}_{2,f_2}\} \quad \text{with} \\
 & & \quad \tilde{F}_{1,j} = (F_{1,j} \times S_2) \cap S_3, \\
 & & \quad \tilde{F}_{2,j} = (S_1 \times F_{2,j}) \cap S_3
\end{array}
$$

Reduction to single fairness constraint Some algorithms operate on Kripke structures with a single fairness constraint. We give two well-known reductions from arbitrary Kripke structures to ones with only one fairness constraint. The process is called *degeneralization*. Let $K = (S, T, I, L, F)$ with $F = \{F_0, \ldots, F_f\} \neq \emptyset$ be a Kripke structure.

The first reduction is referred to as *Choueka's flag construction* [Cho74], see also [Hol03]. It uses $f + 1$ copies of S. Intuitively, a path is forced to cycle through the copies, switching from copy i to copy $(i+1) mod (f+1)$ when a state in F_i is seen. It is now sufficient to define the set of initial states to be the initial states in one of the copies of K and the fair states as the states in F_i in the i-th copy for some i. Formally: we construct $\tilde{K} = (\tilde{S}, \tilde{T}, \tilde{I}, \tilde{L}, \tilde{F})$ with

$$\begin{array}{lcl}
\tilde{S} & = & S \times \{0, \ldots, f\}, \\
\tilde{T} & = & \{((s,i),(s',i')) \mid (s,s') \in T \wedge i' = (\text{if } s \in F_i \text{ then } (i+1) \bmod (f+1) \text{ else } i)\}, \\
\tilde{I} & = & \{(s,0) \mid s \in I\}, \\
\tilde{L}((s,i)) & = & L(s), \quad \text{and} \\
\tilde{F} & = & \{(s,f) \mid s \in F_f\}
\end{array}$$

It's easy to see that $Lang(K) = Lang(\tilde{K})$.

The previous reduction increases the state space by a factor of $\mathbf{O}(f)$. However, lasso-shaped fair paths may become longer than necessary. This effect can be avoided at the expense of using $\mathbf{O}(f)$ instead of $\mathbf{O}(\log(f))$ additional state bits. We refer to the following approach as *bit-set degeneralization*.

$$\begin{array}{lcl}
\tilde{S} & = & S \times \mathbb{B}^{f+1}, \\
\tilde{T} & = & \{((s, b_0, \ldots, b_f), (s', b'_0, \ldots, b'_f)) \mid \\
& & \quad (s,s') \in T \wedge \\
& & \quad (\text{if } (\bigvee_{i=0}^{f} \neg b_i) \text{ then } \forall 0 \le i \le f \,.\, (b_i \rightarrow b'_i) \wedge (b'_i \rightarrow b_i \vee s' \in F_i) \\
& & \qquad\qquad\qquad\qquad \text{else } \forall 0 \le i \le f \,.\, b'_i \rightarrow s' \in F_i)\}, \\
\tilde{I} & = & \{(s, b_0, \ldots, b_f) \mid s \in I \wedge \forall 0 \le i \le f \,.\, b_i \rightarrow s \in F_i\}, \\
\tilde{L}((s, b_0, \ldots, b_f)) & = & L(s), \quad \text{and} \\
\tilde{F} & = & \{(s, 1, \ldots, 1)\}
\end{array}$$

Again, we have $Lang(K) = Lang(\tilde{K})$.¶

2.4 Linear Temporal Logic

Syntax We consider specifications given in Propositional LTL with both future and past operators (PLTLB) [Eme90]. The *syntax* of PLTLB is defined over a set of atomic propositions AP. Each atomic proposition is a PLTLB formula; if ϕ and ψ are PLTLB formulae, so are $\neg\phi$ (*negation*), $\phi \vee \psi$ (*disjunction*), $\mathbf{X}\phi$ (*next-time*), $\phi \ \mathbf{U} \ \psi$ (*strong until*), $\mathbf{Y}\phi$ (*strong last-time*), $\phi \ \mathbf{S} \ \psi$ (*strong since*).

Semantics We define the *semantics* of formulae recursively on positions of infinite sequences over 2^{AP} in Fig. 2.1. The *language* of a formula ϕ is the set of infinite sequences σ such that ϕ holds on σ: $Lang(\phi) = \{\sigma \in (2^{AP})^\omega \mid \sigma, 0 \models \phi\}$. Hence, a PLTLB formula induces a property over AP and the definitions of satisfaction in a Kripke structure from Sect. 2.3 apply. We identify the formula with the property it defines and we write $K \models_{\forall/\exists} \phi$ also for a PLTLB formula ϕ. Via L a formula ϕ also induces a satisfaction relation on paths of a Kripke structure. We write $K, \pi, i \models \phi$ iff $L(\pi[i, \infty]) \models \phi$. If K is clear, it may be omitted; 0 is the default value for i. If only initialized fair paths are taken into account, this provides an alternative route to the definition of existential and universal validity in a Kripke structure. While the latter is preferred by some authors, both definitions are equivalent and ours turns out to make some of the definitions in Sect. 2.7 easier.

¶Note, that forcing b_i to true as soon as a fair state is seen —i.e., using bi-implications $b'_i \leftrightarrow b_i \vee s' \in F_i$, $b'_i \leftrightarrow s' \in F_i$, and $b_i \leftrightarrow s \in F_i$ in $\tilde{T}$ and $\tilde{I}$ —may not guarantee shortest counterexamples. For an example see Fig. 2.3. The problem does not arise if the b_i are set only when the loop has already started.

$$
\begin{array}{lll}
\sigma, i \models p & \text{iff} & p \in \sigma[i] \ \text{ for } \ p \in AP \\
\sigma, i \models \neg\phi & \text{iff} & \sigma, i \not\models \phi \\
\sigma, i \models \phi \vee \psi & \text{iff} & \sigma, i \models \phi \ \text{ or } \ \sigma, i \models \psi \\
\sigma, i \models \mathbf{X}\phi & \text{iff} & \sigma, i+1 \models \phi \\
\sigma, i \models \phi \ \mathbf{U} \ \psi & \text{iff} & \exists j \geq i \ . \ (\sigma, j \models \psi \ \wedge \ \forall i \leq k < j \ . \ \sigma, k \models \phi) \\
\sigma, i \models \mathbf{Y}\phi & \text{iff} & i > 0 \ \text{ and } \ \sigma, i-1 \models \phi \\
\sigma, i \models \phi \ \mathbf{S} \ \psi & \text{iff} & \exists 0 \leq j \leq i \ . \ (\sigma, j \models \psi \ \wedge \ \forall j < k \leq i \ . \ \sigma, k \models \phi)
\end{array}
$$

Figure 2.1: The semantics of PLTLB

$$
\begin{array}{lll}
sub(p) & = & \{p\} \\
sub(\neg\phi) & = & \{\neg\phi\} \cup sub(\phi) \\
sub(\phi \vee \psi) & = & \{\phi \vee \psi\} \cup sub(\phi) \cup sub(\psi) \\
sub(\mathbf{X}\phi) & = & \{\mathbf{X}\phi\} \cup sub(\phi) \\
sub(\phi \ \mathbf{U} \ \psi) & = & \{\phi \ \mathbf{U} \ \psi\} \cup sub(\phi) \cup sub(\psi) \\
sub(\mathbf{Y}\phi) & = & \{\mathbf{Y}\phi\} \cup sub(\phi) \\
sub(\phi \ \mathbf{S} \ \psi) & = & \{\phi \ \mathbf{S} \ \psi\} \cup sub(\phi) \cup sub(\psi)
\end{array}
$$

Table 2.1: Definition of subformulae

Future and past fragments If the past operators $\mathbf{Y}$ and $\mathbf{S}$ are excluded, we obtain *future* LTL formulae (PLTLF). Similarly, a *past* formula (PLTLP) has no occurrences of $\mathbf{X}$ and $\mathbf{U}$. For this reason, when we speak about future or past, we include present.

Further operators We have the following usual abbreviations: $1 \equiv p \vee \neg p$ (*constant true*), $0 \equiv \neg 1$ (*constant false*), $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$ (*conjunction*), $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$ (*implication*), $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ (*equivalence*), $\phi \ \mathbf{R} \ \psi \equiv \neg(\neg\phi \ \mathbf{U} \ \neg\psi)$ (*release*), $\mathbf{F}\phi \equiv 1 \ \mathbf{U} \ \phi$ (*finally*), $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$ (*globally*), $\mathbf{Z}\phi \equiv \neg\mathbf{Y}\neg\phi$ (*weak last-time*), $\phi \ \mathbf{T} \ \psi \equiv \neg(\neg\phi \ \mathbf{S} \ \neg\psi)$ (*weak since*, "triggered"), $\mathbf{O}\phi \equiv 1 \ \mathbf{S} \ \phi$ (*once*), and $\mathbf{H}\phi \equiv \neg\mathbf{O}\neg\phi$ (*historically*).

Recursive expansion formulae For $\mathbf{U}$ and $\mathbf{S}$ there exist recursive expansion formulae (e.g. [KPR98]):

$$
\begin{array}{lllll}
\phi = \psi_1 \ \mathbf{U} \ \psi_2 & : & \pi_i \models \phi & \text{iff} & (\pi_i \models \psi_2) \vee (\pi_i \models \psi_1) \wedge (\pi_{i+1} \models \phi) \\
\phi = \psi_1 \ \mathbf{S} \ \psi_2 & : & \pi_i \models \phi & \text{iff} & (\pi_i \models \psi_2) \vee (i > 0) \wedge (\pi_i \models \psi_1) \wedge (\pi_{i-1} \models \phi)
\end{array}
$$

The expansion of $\mathbf{U}$ is not sufficient to guarantee proper semantics: additional measures must be taken to select the desired fixed point, e.g., by adding fairness constraints.

Sub-/superformula The set of *subformulae* of a PLTLB formula ϕ, $sub(\phi)$, is defined recursively in Tab. 2.1. Further, if $\phi \in sub(\psi)$, then ψ is a *superformula* of ϕ.

Future/past operator depth The *future (resp. past) operator depth* h_f (resp. h_p) of a PLTLB formula ϕ is the maximal number of nested future (resp. past) time operators in ϕ:

$$h_f(\phi) = \begin{cases} 0 & \text{iff} \quad \phi \in AP \\ h_f(\psi) & \text{iff} \quad \phi = \circ\psi \text{ , where } \circ \in \{\neg, \mathbf{Y}\} \\ \max(h_f(\psi_1), h_f(\psi_2)) & \text{iff} \quad \phi = \psi_1 \circ \psi_2 \text{ , where } \circ \in \{\vee, \mathbf{S}\} \\ 1 + h_f(\psi) & \text{iff} \quad \phi = \mathbf{X}\psi \\ 1 + \max(h_f(\psi_1), h_f(\psi_2)) & \text{iff} \quad \phi = \psi_1 \ \mathbf{U} \ \psi_2 \end{cases}$$

$$h_p(\phi) = \begin{cases} 0 & \text{iff} \quad \phi \in AP \\ h_p(\psi) & \text{iff} \quad \phi = \circ\psi \text{ , where } \circ \in \{\neg, \mathbf{X}\} \\ max(h_p(\psi_1), h_p(\psi_2)) & \text{iff} \quad \phi = \psi_1 \circ \psi_2 \text{ , where } \circ \in \{\vee, \mathbf{U}\} \\ 1 + h_p(\psi) & \text{iff} \quad \phi = \mathbf{Y}\psi \\ 1 + max(h_p(\psi_1), h_p(\psi_2)) & \text{iff} \quad \phi = \psi_1 \ \mathbf{S} \ \psi_2 \end{cases}$$

The authors of [LMS02, BC03] proved independently that a PLTLB property ϕ can distinguish at most $h_p(\phi)$ loop iterations of a lasso. We restate Lemma 5.2 of [LMS02] for PLTLB:

Lemma 4 *For any lasso π of type (l_s, l_l), for any PLTLB property ϕ with at most $h_p(\phi)$ nested past-time modalities, and any $i \geq l_s + l_l \cdot h_p(\phi)$: $\pi, i \models \phi \Leftrightarrow \pi, i + l_l \models \phi$.*

2.5 Büchi Automata

Büchi automata as operational representations Linear temporal logic formulae are descriptive in nature and, given some experience, are relatively easy to read and write. Büchi automata [Büc62, Tho90] are one kind of finite state automata whose acceptance condition is designed to accept infinite words (sequences) over some alphabet. They have a close relation to PLTLB (see below) and they have an operational character. Therefore, automata on infinite objects [Tho90] such as Büchi automata are a working horse of many model checking algorithms.

Definition Given our definition of fair Kripke structures, a Büchi automaton that accepts infinite sequences over 2^{AP} is simply a fair Kripke structure.[‖] The *language* accepted by the Büchi automaton $B = (S, T, I, L, F)$ is $Lang(B)$. An initialized fair path ρ in B with $L(\rho) = \alpha$ is a *run* on α. A Büchi automaton is called *generalized* if $|F| > 1$, non-generalized or simple otherwise. Every generalized Büchi automaton can be transformed into a non-generalized one accepting the same language (see Sect. 2.3).

Union, intersection, complement Typical operations on formal languages and the automata used to accept them are union, intersection, and complement. Computing language *union* (resp. *intersection*) for a pair of Büchi automata B_1, B_2 can be achieved by forming their sum (resp. synchronous product) $B_1 + B_2$ (resp. $B_1 \times B_2$). While Büchi automata are closed under complement, the construction is exponential in the size of the automaton and usually avoided if possible (see [Tho90] for references).

[‖]Typically, automata used as language acceptors are labeled with the their alphabet on transitions rather than states. However, model checking algorithms are formulated easier using this definition; see also [Pel01].

Emptiness, inclusion Below we will need to determine whether the language accepted by a Büchi automaton B is empty or not. The language of $B = (S, T, I, L, F)$ is *not empty* iff there exists an initialized fair path, or, equivalently, there exists a reachable, fair, non-trivial strongly connected component. The latter can be determined in linear time in the number of states of the automaton using Tarjan's algorithm [Tar72] or nested depth-first search [CVWY92]. Testing language inclusion between two Büchi automata is PSPACE-hard in general [CDK93].

Relating ω-regular expressions, PLTLB, and Büchi automata It turns out that there is an intimate relationship between the languages that can be expressed using ω-regular languages, PLTLB, and Büchi automata. It is this relationship that makes use of Büchi automata in model checking linear time properties worthwhile. ω-regular expressions and Büchi automata are expressively equivalent. PLTLB has less expressive power than the other two formalisms: its expressivity corresponds to that of star-free ω-regular expressions and of counter-free Büchi automata. Extended versions of linear temporal logic such as [Wol83, SVW87, VW94] raise the expressive power of linear temporal logic to that of ω-regular expressions and of Büchi automata. For more explanation, proofs, and references see, e.g., [Eme90, Tho90].

2.6 Translating PLTLB Formulae into Büchi Automata

Motivation Leveraging the operational character of Büchi automata for verification of PLTLB formulae requires a translation from a PLTLB formula ϕ into a Büchi automaton B^ϕ such that $Lang(\phi) = Lang(B^\phi)$. Wolper, Vardi, and Sistla were the first to show that this is possible and to provide a corresponding algorithm [WVS83, VW94]. Below we follow the construction of Kesten et al. [KPR98].**

Construction A Büchi automaton B^ϕ_{KPR} for a PLTLB formula ϕ is constructed symbolically as $B^\phi_{KPR} = (V^\phi, S^\phi, T^\phi, I^\phi \wedge x_\phi, L^\phi, F^\phi)$ over the set of atomic propositions $AP^\phi = \{p \mid p$ is an atomic proposition in $sub(\phi)\}$. All state variables are Boolean, V^ϕ, S^ϕ, T^ϕ, I^ϕ, and F^ϕ are recursively defined in Tab. 2.2, and $L^\phi(s) = \{p \in AP^\phi \mid x_p(s) = 1\}$. For a uniform explanation, Tab. 2.2 uses state variables also for Boolean connectives. In an implementation for BDDs these are typically replaced by macros [KPR98, CGH97, Sch01]. Intuitively, each state variable of B^ϕ_{KPR} corresponds to a subformula ψ of ϕ. x_ψ is constrained such that x_ψ is true at some state of an initialized fair path π in B^ϕ_{KPR} iff ψ is true in that state: $\forall \pi \in \Pi^F \,.\, \forall \psi \in sub(\phi) \,.\, \forall 0 \leq i \,.\, x_\psi(\pi[i]) \Leftrightarrow \pi, i \models \psi$. For that purpose, the definitions of the Boolean connectives and $\mathbf{X}$ and $\mathbf{Y}$ in Tab. 2.2 directly follow the semantics of PLTLB in Fig. 2.1. For $\mathbf{U}$ and $\mathbf{S}$ the recursive expansion formulae are used. The fairness properties ensure that the right subformula of an $\mathbf{U}$ eventually becomes true. For more explanations and formal proofs of correctness see [KPR98, CGH97, LPZ85].

Complexity and application Note that although B^ϕ_{KPR} has exponentially many states in $|\phi|$, a symbolic description has length $\mathbf{O}(|\phi|)$ and can be constructed in $\mathbf{O}(|\phi|)$ time and space. Hence, the construction is often used in symbolic model checking (which is the focus of our

**Similar constructions have appeared before but their presentation is closest to our definitions.

ψ	definition				
	$V^\psi =$	$S^\psi =$	$T^\psi =$	$I^\psi =$	$F^\psi =$
p	$\{x_p\}$	$x_p \leftrightarrow p$	1	1	$\emptyset$
$\neg\psi_1$	$V^{\psi_1} \cup \{x_\psi\}$	$S^{\psi_1} \wedge (x_\psi \leftrightarrow \neg x_{\psi_1})$	T^{ψ_1}	I^{ψ_1}	F^{ψ_1}
$\psi_1 \vee \psi_2$	$V^{\psi_1} \cup V^{\psi_2} \cup \{x_\psi\}$	$S^{\psi_1} \wedge S^{\psi_2} \wedge$ $(x_\psi \leftrightarrow x_{\psi_1} \vee x_{\psi_2})$	$T^{\psi_1} \wedge T^{\psi_2}$	$I^{\psi_1} \wedge I^{\psi_2}$	$F^{\psi_1} \cup F^{\psi_2}$
$\mathbf{X}\psi_1$	$V^{\psi_1} \cup \{x_\psi\}$	S^{ψ_1}	$T^{\psi_1} \wedge (x_\psi \leftrightarrow x'_{\psi_1})$	I^{ψ_1}	F^{ψ_1}
$\psi_1 \ \mathbf{U} \ \psi_2$	$V^{\psi_1} \cup V^{\psi_2} \cup \{x_\psi\}$	$S^{\psi_1} \wedge S^{\psi_2}$	$T^{\psi_1} \wedge T^{\psi_2} \wedge$ $(x_\psi \leftrightarrow x_{\psi_2} \vee x_{\psi_1} \wedge x'_\psi)$	$I^{\psi_1} \wedge I^{\psi_2}$	$F^{\psi_1} \cup F^{\psi_2} \cup$ $\{\{\neg x_\psi \vee x_{\psi_2}\}\}$
$\mathbf{Y}\psi_1$	$V^{\psi_1} \cup \{x_\psi\}$	S^{ψ_1}	$T^{\psi_1} \wedge (x'_\psi \leftrightarrow x_{\psi_1})$	$I^{\psi_1} \wedge (x_\psi \leftrightarrow 0)$	F^{ψ_1}
$\psi_1 \ \mathbf{S} \ \psi_2$	$V^{\psi_1} \cup V^{\psi_2} \cup \{x_\psi\}$	$S^{\psi_1} \wedge S^{\psi_2}$	$T^{\psi_1} \wedge T^{\psi_2} \wedge$ $(x'_\psi \leftrightarrow x'_{\psi_2} \vee x'_{\psi_1} \wedge x_\psi)$	$I^{\psi_1} \wedge I^{\psi_2} \wedge$ $(x_\psi \leftrightarrow x_{\psi_2})$	$F^{\psi_1} \cup F^{\psi_2}$

Table 2.2: Property-dependent part of a Büchi automaton constructed with KPR [KPR98]

work), while in explicit state model checking constructions that produce smaller automata are preferred. For more discussion and references see Sect. 5.5.

2.7 Defining Safety and Liveness

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

Douglas Adams, Mostly Harmless

All living things are afraid to die. — No, you're exactly wrong, the only truly alive beings are those unafraid to die.

David Zindell, Neverness

Formal definition of safety Alpern and Schneider [AS85] were the first to formally characterize both, safety and liveness properties (for an informal explanation see Sect. 1.1). The basis of their definition of a safety property is the idea that if a "bad thing" has happened, that must be irremediable. Hence, a property ϕ over 2^{AP} is a safety property iff each sequence not in ϕ has a finite *bad prefix*, i.e., a finite prefix that cannot be extended to a sequence in ϕ:

$$\phi \quad \text{is a safety property} \quad \text{iff} \quad (\forall \sigma \ . \ \sigma \notin \phi) \Leftrightarrow (\exists 0 \leq i \ . \ \forall \tau \in (2^{AP})^\omega \ . \ \sigma[0..i] \circ \tau \notin \phi)$$

Note that, by this definition, the "bad thing" must be discrete and there is an identifiable point at which it can be recognized [AS85]. Now it is easy to see why bounded liveness properties are actually safety properties: passing of the bound without the "good thing" having happened constitutes a bad thing and is beyond remedy.

Formal definition of liveness Key to Alpern and Schneider's definition of a liveness property is the idea that every finite prefix can be extended to a sequence in ϕ — otherwise that finite prefix would be a bad prefix and, hence, the property would also have at least partial characteristics of a safety property. Formally:

$$\phi \quad \text{is a liveness property} \quad \text{iff} \quad \forall \sigma \in (2^{AP})^* \ . \ \exists \tau \in (2^{AP})^\omega \ . \ \sigma \circ \tau \in \phi$$

Note that the "good thing" need not be discrete.

Justification and consequences Alpern and Schneider argue [AS85] that no definition of a liveness property can be more permissive than theirs: any property failing the above definition must have a finite prefix that cannot be extended to be in the property; however, this is by definition a bad prefix. They further give a topological interpretation where safety properties are closed sets and liveness properties are dense sets. Based on that they show that every property is the intersection of a safety and a liveness property. The only property that is both a safety and a liveness property is the true property $(2^{AP})^{\omega}$. In a subsequent paper [AS87] the same authors give an equivalent characterization in terms of Büchi automata. It allows to check whether a property given as Büchi automaton is a safety or a liveness property. Furthermore, they show how to construct Büchi automata corresponding to a safety and liveness property whose intersection is the original property. The definition of Alpern and Schneider has been widely adopted (e.g., [KV01, Pel01, Hol03]), refined into a temporal hierarchy [MP90], and generalized using a lattice-theoretic approach to encompass branching time [MT03] and using Heyting algebras to handle finite behaviors [Mai04].

Alternative definitions Lamport provided the first formal definition of safety [Lam85]. His definition is based on the idea that for a safety property ϕ, an infinite path satisfies ϕ iff each finite prefix does not violate ϕ. He restricts his definition to stuttering-invariant [Lam83] properties; other than that, the definition is equivalent to that of Alpern and Schneider [AS85], see [ADS86]. The definition [AS85] of safety properties coincides with Emerson's limit closed properties [Eme83]. Lichtenstein et al. provide a syntactic definition of safety properties using past operators [LPZ85], which is equivalent to [AS85]: every formula of the form $\mathbf{G}\phi$, where ϕ is a past formula, describes a safety property. They also give a syntactic definition of liveness properties, which is, however, more general than [AS85]. Sistla [Sis94] gives syntactic characterizations of several classes of safety properties using only future operators, the most general being equivalent to [AS85] for properties that can be expressed in PLTLB. He also syntactically characterizes some classes of liveness and fairness properties. For a (somewhat dated) survey see [Kin94].

Recognizing bad prefixes Kupferman and Vardi investigated how the knowledge that a property is a safety property can be used in verification of that property [KV01]. According to our definition (Sect. 2.3), a counterexample to some property ϕ in a model M is an infinite path π in M that violates ϕ. If ϕ is a safety property, a (finite) bad prefix of a counterexample focuses on the violating part and, hence, should be more useful to the developer. Kupferman and Vardi show how to construct an automaton on finite words, which accepts precisely the set of shortest bad prefixes for a safety property ϕ. They call such an automaton *tight* for ϕ. The construction involves an exponential blowup when starting from a Büchi automaton accepting $\neg\phi$ and a corresponding double exponential blowup when ϕ is given as PLTLB formula. In the light of that blowup the requirement to accept every violating prefix is lessened: an automaton on finite words is *fine* for ϕ if it accepts at least one bad prefix for each $\pi \notin \phi$.

Kupferman and Vardi then introduce the notion of an *informative* bad prefix. Intuitively, an informative bad prefix contains all the information to see why the prefix is bad for a safety formula ϕ, it "tells the whole story" [KV01]. The idea of the formal definition of that notion is that the satisfaction of the formula $\neg\phi$ on a bad prefix π can be established by proceeding as in Fig. 2.1 without ever having to refer beyond the last state of π. Safety formulae are then classified as *intentionally safe* iff every bad prefix is informative, as *accidentally safe* iff every

$\pi \notin \phi$ has at least one informative bad prefix, and as *pathologically safe* otherwise. The latter contain redundancy and are not expected to occur often in practice. There exists an automaton B on finite words exponential in the size of a PLTLB formula ϕ such that B is tight for ϕ if ϕ is intentionally safe and fine for ϕ if ϕ is accidentally safe.

2.8 Model Checking Linear Time Properties

2.8.1 Basics

Model checking problem Given a model of a system M as Kripke structure and a property ϕ as PLTLB formula or as Büchi automaton, both over a set of atomic propositions AP, the model checking problem is to determine whether $M \models_\forall \phi$.

Automata-theoretic approach for linear time We have already seen that Büchi automata can represent both, model and PLTLB formula. Let M be a model and ϕ be a PLTLB formula over a common set of atomic propositions AP, and let B^ϕ, $\overline{B^\phi}$, and $B^{\neg\phi}$ be Büchi automata where B^ϕ accepts ϕ, $\overline{B^\phi}$ is the complement of B^ϕ, and $B^{\neg\phi}$ accepts $\neg\phi$. This leaves the following choices to solve the problem whether $M \models_\forall \phi$: $Lang(M) \overset{?}{\subseteq} Lang(B^\phi)$, $Lang(M \times \overline{B^\phi}) \overset{?}{=} \emptyset$, or $Lang(M \times B^{\neg\phi}) \overset{?}{=} \emptyset$. In the light of the complexities hinted at in Sect. 2.5 the last choice is preferable. Model checking for a model M and a PLTLB formula ϕ can be done in time $\mathbf{O}(|M| \cdot 2^{|\phi|})$, where $|M|$ is the number of states in M and $|\phi|$ is the length of ϕ, as follows [VW86]:

1. negate ϕ: $\mathbf{O}(1)$
2. construct $B^{\neg\phi}$: $\mathbf{O}(2^{|\phi|})$
3. construct $M \times B^{\neg\phi}$: $\mathbf{O}(|M| \cdot |B^{\neg\phi}|)$
4. check whether $Lang(M \times B^{\neg\phi}) = \emptyset$: $\mathbf{O}(|M| \cdot |B^{\neg\phi}|)$

If the product automaton has only one fairness constraint, step 4 corresponds to checking whether there is a fair state that is both, reachable from an initial state and reachable from itself. Such a state is termed *repeatedly reachable*.

Model checking safety properties As noted in Sect. 2.7, for each safety property ϕ there exists an automaton on finite words that recognizes the bad prefixes of ϕ. This allows for a simpler procedure to model check safety properties [KV01]: Transform the Büchi automaton representing the model into an automaton on finite words by disregarding the set of fairness constraints and making every state accepting. Then build the product with the automaton recognizing the bad prefixes. Finally, determine whether an accepting state (also called a *bad state*) in the product is *reachable*.

Reachability and repeated reachability The last two paragraphs justify the notion that safety properties can be checked by reachability while general linear properties require repeated reachability.

2.8.2 Lasso-shaped counterexamples

Existence of lasso-shaped counterexamples From the automata-theoretic approach it's easy to see that, if a counterexample π' to a PLTLB property ϕ exists in a model M given as Kripke structure, then there also exists a lasso-shaped counterexample π to ϕ in M [VW86].[††] The length of a lasso-shaped counterexample π is defined as the length of its minimal lasso.

Shortest counterexamples Given that

1. we are only interested in finitely representable counterexamples,
2. every failing PLTLB property in a given Kripke structure M has a lasso-shaped counterexample, and
3. lasso-shaped counterexamples are returned by most model checking algorithms for PLTLB,

we adopt the following definition from Clarke et al. [CGMZ95]: a shortest counterexample to a PLTLB property ϕ in a Kripke structure M is one that has a most compact representation as a lasso. Formally, let $M = (S, T, I, L, F)$ be a Kripke structure, let ϕ be a PLTLB property. A path α in M is a *shortest counterexample* for ϕ in M iff

1. $\alpha \not\models \phi$
2. $\exists \beta, \gamma \;.\; (\alpha = \beta\gamma^\omega \wedge \forall \beta', \gamma' \;.\; (\beta'\gamma'^\omega \in \Pi^F \wedge \beta'\gamma'^\omega \not\models \phi \Rightarrow |\langle \beta, \gamma \rangle| \leq |\langle \beta', \gamma' \rangle|))$

Discussion This definition is not optimal. First, an early position of the violation (if that can be clearly attributed) need not coincide with the least number of states required to close a loop. Second, apart from length, ease of understanding is not a criterion either. The first problem is most relevant for properties that also have finite bad prefixes, i.e., properties that are a subset of a safety property [KV01]. Finding the shortest bad prefix for safety formulae can be done using the (doubly exponential) method proposed in [KV01]. For solutions to the second problem see the discussion of related work in Chap. 3.

2.8.3 Model checking using BDDs

ROBDDs Binary decision diagrams [Bry86] represent a Boolean function as directed acyclic graph where internal nodes are marked with variables, outgoing edges are marked with potential values 0 and 1 of a variable, and terminal nodes contain the result 0 or 1 of a function application. If common subgraphs are shared, the BDD is *reduced*. It is *ordered*, if the sequence of nodes in all paths from a root to a leaf follows the same variable order. We assume reduced ordered BDDs (ROBDDs) with a common variable order for all BDDs involved. For a given variable order, ROBDDs are a canonical normal form for Boolean functions. Operations include Boolean operations, (partial) function evaluation, and function composition.

[††] Sistla and Clarke proved that fact before [SC85].

Importance of variable order The actual size of a BDD representing a given function and, hence, the time needed to perform operations involving that BDD, depend to a large extent on the variable order used. The difference in size between an optimal and a non-optimal variable order may be exponential [Bry86]. Finding an optimal variable order is a coNP-complete problem [Bry86]. For some Boolean functions the smallest BDD has size exponential in the number of variables [Bry91].

Reachability and repeated reachability with BDDs BDD-based symbolic model checking represents both, transition relation and sets of states (more precisely, their characteristic functions), as BDDs. Given a set of states S_1 as BDD one can compute BDDs representing the sets of states reachable from S_1 in one step forward (*forward image*) or backward (*backward image*). The set of reachable states can then simply be constructed by starting from the initial states and repeatedly computing forward images until a fixed point is reached. This corresponds to a breadth-first traversal of the state graph. A check whether a bad state has been reached can be performed after each iteration. The number of forward image operations to determine reachability of a bad state is the minimum of the radius of the state graph and the distance of the bad state closest to the initial states. Repeated reachability is more involved, it requires two or more fixed point computations.

Forward model checking The standard method to evaluate CTL formulae in a BDD-based model checker is based on backward image computation [McM93]. Experimental evidence shows that computing forward images performs better than computing backward images [INH96]. Triggered by that observation Iwashita et al. propose *forward model checking*, which tries to replace backward image computations with forward image computations in the evaluation of a CTL formula [INH96]. Henzinger et al. show that all ω-regular properties can be handled using forward image computations only, while some CTL formulae require backward image computation [HKQ98]. Biere et al. then suggest a symbolic tableau-based method for forward model checking of PLTLF, which combines depth- and breadth-first search [BCZ99]. Forward model checking inherently only traverses the reachable state space; this can also be accomplished with backward image computation by computing the set of reachable states first, but that incurs an additional risk of state space explosion. As stated in the previous paragraph checking reachability requires forward image computation only.

2.8.4 Bounded model checking using SAT solvers

Idea, Process In SAT-based bounded model checking [BCCZ99, BCC+99, BCRZ99], the model checking problem $M \models_\forall \phi$ is translated into a sequence of propositional formulae of the form $|[M, \phi, k]|$ in the following way: $|[M, \phi, k]|$ is satisfiable iff an informative bad prefix or a lasso-shaped counterexample π of length k exist. In the case of a lasso-shaped counterexample, a loop is assumed to be closed between the last state $\pi[k-1]$ and some successor $\pi[l+1]$ of a previous occurrence of that last state $\pi[l] = \pi[k-1]$. The resulting formulae are then handed to a SAT solver for increasing bounds k until either a counterexample is found, absence of a counterexample is proved, or a user defined resource threshold is reached. Note, that checking reachability and repeated reachability is usually combined in SAT-based bounded model checking.

Custom encodings versus Büchi automata In principle, the automata-theoretic approach can be used to encode $|[M, \phi, k]|$; however, first implementations used a custom encoding [BCCZ99, BC03]. This proved difficult to implement in an optimal fashion and to extend for completeness. More recent work uses either simplified, recursive encodings [CRS04, LBHJ04, LBHJ05, HJL05], which are optimized for BMC but show some similarity to the Büchi automata of [KPR98], or uses Büchi automata directly [CKOS05, AS04].

Incremental BMC *Incremental* bounded model checkers, introduced by [Sht01, WKS01] allow the SAT solver to reuse partial results obtained for some bound k_1 when checking $k_2 > k_1$, often giving significant speed-ups. Recent implementations in NuSMV include [ES03] for invariants and [HJL05] for PLTLB.

2.8.5 Abstraction

Existential abstraction To obtain an abstract Kripke structure $\tilde{M} = (\tilde{S}, \tilde{T}, \tilde{I}, \tilde{L}, \tilde{F})$ from some concrete model $M = (S, T, I, L, F)$, abstraction typically introduces a surjective mapping h from the concrete state space S to an abstract set of states $\tilde{S}$ where $\tilde{S}$ has fewer states than S [CGL94]. If the abstraction is *existential*, there is a transition between abstract states $\tilde{s}$, $\tilde{s}'$ if there exist concrete states s, s' such that $h(s) = \tilde{s}$, $h(s') = \tilde{s}'$, and $(s, s') \in T$. Similarly, $\tilde{s}$ is initial (resp. $\in \tilde{F}_i$), if $\tilde{s} = h(s)$ for some initial state s (resp. for some $s \in F_i$). Note that this formulation requires the concrete transition relation to obtain the abstract transition relation. As a representation of the concrete transition relation as a BDD might already require too much memory further approximations are performed to obtain the abstract transition relation directly from a relational description of the original model. Both, the existential abstraction and the (suitably chosen) further approximations only add behavior to the original Kripke structure. Hence, if a property ϕ holds universally in the abstracted and approximated model, it is known to hold universally in the concrete model [CGL94].

Universal abstraction The definition of *universal abstraction* can be obtained by replacing existential with universal quantifiers in the definition of existential abstraction. Universal abstraction only restricts behavior of the original Kripke structure and, therefore, can be used to establish that a property ϕ holds existentially. We do not use universal abstraction in this dissertation and refer to existential abstraction when we speak of abstraction.

Abstraction refinement Only if a property ϕ can be proven to hold universally in the abstract Kripke structure $\tilde{M}$ the result directly transfers to the concrete M. If ϕ does not hold universally in the abstract, a concrete counterexample can sometimes be reconstructed from an abstract one and a definite result is obtained for M. Otherwise the abstraction needs to be refined. Figure 2.2 shows that scheme, which has been proposed in similar form by Balarin and Sangiovanni-Vincentelli [BSV93] and Kurshan [Kur94]. The algorithm terminates provided that a strict refinement can always be obtained in line 9 and the maximal number of successive refinements in line 9 is finitely bounded (and, of course, “elementary” steps in lines 1, 3, and 6 terminate).

Require: concrete model M and property ϕ
Ensure: return 1 iff $M \models_\forall \phi$
1: construct initial abstraction $\tilde{M}$, $\tilde{\phi}$
2: **loop**
3:    model check $\tilde{M} \overset{?}{\models}_\forall \tilde{\phi}$
4:    **if** $\tilde{M} \models_\forall \tilde{\phi}$ **then**
5:       return 1
6:    **else if** counterexample has concrete counterpart **then**
7:       return 0
8:    **else**
9:       refine $\tilde{M}$, $\tilde{\phi}$ based on information obtained in line 3
10:    **end if**
11: **end loop**

Figure 2.2: A general scheme for abstraction refinement

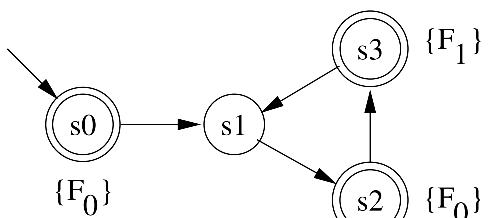


Figure 2.3: This example shows that the variant of bit-set degeneralization that forces b_i to true as soon as a state in F_i is seen may not give shortest counterexamples. The example has two sets of fairness constraints F_0 and F_1. If b_0 is forced to true in the initial state s_0, a loop in the degeneralized automaton can only be closed when s_2 is seen for the second time. Specifically, the resulting fair lasso is $(s_0 s_1)(s_2 s_3 s_1)^\omega$, while the shortest fair lasso is $(s_0)(s_1 s_2 s_3)^\omega$.

3

Symbolic Loop Detection for Finite State Systems

If you try and take a cat apart to see how it works, the first thing you have on your hands is a non-working cat. Life is a level of complexity that almost lies outside our vision; it is so far beyond anything we have any means of understanding that we just think of it as a different class of object, a different class of matter; 'life', something that had a mysterious essence about it, was god given, and that's the only explanation we had.

Douglas Adams

In this chapter we present the state-recording translation from repeated reachability to reachability, which is the main idea of this dissertation. Section 3.1 introduces a translation from simple liveness to safety. This is extended to fair repeated reachability and formalized in 3.2. Its complexity is analyzed in Sect. 3.3. Section 3.4 explains how shortest lasso-shaped counterexamples can be found. Section 3.5 discusses related work and Sect. 3.6 sums up.

3.1 Translating Simple Liveness into Safety

3.1.1 Intuition

Lasso-shaped counterexamples A counterexample trace for a simple liveness property $\mathbf{F}p$ is an infinite path where p never holds along the path. If the number of states in a system is finite, a counterexample trace to a simple liveness property can be assumed to be lasso-shaped: it consists of a finite prefix and an infinitely repeating loop as shown in Fig. 3.1 (see also Sect. 2.8.2). Such a trace can always be derived from an arbitrary infinite trace by inserting a back loop from the first state occurring the second time. If p was false for every state in the original trace, it will also hold nowhere in the lasso-shaped trace.

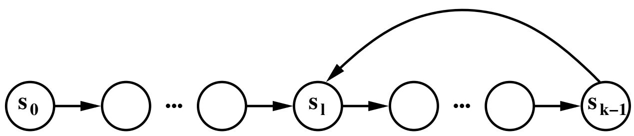


Figure 3.1: A generic lasso-shaped counterexample

Special-purpose algorithms Thus, simple liveness properties $\mathbf{F}p$ of finite state systems can be verified by finding all lasso-shaped traces and checking whether p has been true somewhere on each trace once the loop is closed. Explicit state algorithms using Büchi Automata [CVWY92] and unfolding liveness properties in SAT-based symbolic bounded model checking [BCCZ99] are examples of model checking algorithms that use this observation. Instead of implementing this observation in a special purpose algorithm we show in the following how it can be used to transform a model and a simple liveness property such that reachability checking is sufficient to verify that property.

Translating bounded liveness In model checking applications it is often observed that a simple liveness property $\mathbf{F}p$ can further be restricted by adding a bound k on the number of steps within which the body p has to hold. The bound is either given in the specification or may be determined by manual inspection. A bounded simple liveness property $\mathbf{F}^k p$ is defined as

$$\mathbf{F}^k p \equiv p \vee \mathbf{X}p \vee \cdots \vee \mathbf{X}^k p, \text{ with } \mathbf{X}^i p \equiv \underbrace{\mathbf{X} \cdots \mathbf{X}}_{i-\text{times}} p \tag{3.1}$$

and clearly $\mathbf{F}^k p$ implies $\mathbf{F}p$. The reverse direction is also true if the bound is chosen large enough, in particular as large as the number of states $|S|$ in the model, since all states are reachable in $|S|$ steps. A naive translation would just exchange $\mathbf{F}p$ for $\mathbf{F}^k p$ with k the number of states. However, the expansion of $\mathbf{F}^k p$ in (3.1) results in a very large formula, especially in the context of symbolic model checking.

Translating unbounded liveness Assume instead, that the model is extended with a variable *looped* that indicates when a loop is closed and with a variable *live* that remembers whether p has already been true. Then, the simple liveness property $\mathbf{F}p$ in the original model is equivalent to the safety property $\mathbf{G}(\textit{looped} \rightarrow \textit{live})$ in the extended model. Implementing *live* is easy. In the rest of this section two implementations for *looped* are discussed. The first *counter-based translation* is based on the verification of bounded liveness alone as described above. A main contribution of this dissertation is the second *state-recording translation*, which can be applied to arbitrary finite state systems and ω-regular properties and can still be verified efficiently in many cases.

Example As an example, consider the 2-bit counter with self-loops in Fig. 3.2. There, $\mathbf{F}\ s = 3$ does not hold. A counterexample is given by $\pi = 0, 1, 2, 2, \ldots$. Figure 3.3 shows a model of the counter in the input language of the model checker *NuSMV* [CCG⁺02] in its original form and with the counter-based and the state-recording translation applied. Note that all three models explicitly enumerate all possible values of the counter. While this makes the description easier to understand, it is exponential in the number of bits of the counter. A linear description can be obtained by using a binary encoding of *s* in the declaration of the variables and in the transition relation.

Remarks Note, that in Fig. 3.3 only the specifications are not supported by the input language of the original SMV [McM93], all other parts are compatible. Our reduction can be used in every model checker that supports verifying invariants. In the original SMV these are expressed as $\mathbf{AG}\ \textit{invariant}$ and, hence, we could have written $\mathbf{AG}\ (\textit{looped} \rightarrow \textit{live})$. We use the dialect to

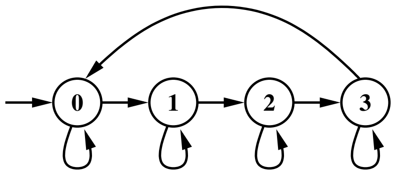


Figure 3.2: A 2-bit counter with self-loops

```
MODULE main



VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1,s};
    s = 1: {2,s};
    s = 2: {3,s};
    s = 3: {0,s};
  esac;

LTLSPEC
  F (s = 3)
```

(a) original

```
MODULE main

-- unmodified part of the
-- original system
VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1,s};
    s = 1: {2,s};
    s = 2: {3,s};
    s = 3: {0,s};
  esac;

-- lasso detection part
VAR
  counter: 0..4;
  live   :  boolean;

ASSIGN
  init(counter) := 0;
  next(counter) := case
    counter < 4: counter + 1;
    1          : counter;
  esac;

  init(live) := 0;
  next(live) := live | (s = 3);

DEFINE
  looped := (counter = 4);

-- transformed specification
INVARSPEC
  looped -> live;
```

(b) counter-based

```
MODULE main

-- unmodified part of the
-- original system
VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1,s};
    s = 1: {2,s};
    s = 2: {3,s};
    s = 3: {0,s};
  esac;

-- lasso detection part
VAR
  save : boolean;
  hat_s: {0, 1, 2, 3};
  lo   : {st,lb,lc};
  live : boolean;

ASSIGN
  init(lo) := st;
  next(lo) := case
    (lo = st) & save       : lb;
    (lo = lb) & (hat_s = s): lc;
    1                      : lo;
  esac;

  init(hat_s) := 0;
  next(hat_s) := case
    (lo = st) & save: s;
    1               : hat_s;
  esac;

  init(live) := 0;
  next(live) := live | (s = 3);

DEFINE
  looped := (lo = lc);

-- transformed specification
INVARSPEC
  looped -> live
```

(c) state-recording

Figure 3.3: Translating simple liveness: NuSMV code of 2-bit counter with self-loops

emphasize the difference between the original (PLTLB) and transformed (invariant) versions. Another reason is to stick to the linear view throughout this dissertation. Finally, while the PLTLB property $\mathbf{F}\ s = 3$ is equivalent* to the CTL property $\mathbf{AF}\ s = 3$, we will later also use PLTLB specifications that cannot be expressed in CTL, which is the only property language directly supported by SMV.

3.1.2 Counter-Based Translation

Intuition Instead of detecting a loop when it is closed, the counter-based translation infers that a loop should have occurred once a sufficient number of transitions have been performed. A counter is added to the model that is incremented at each transition and sets *looped* to true once it reaches a predefined bound.

Example In Fig. 3.3 (b) the state variables and the transition relation of the original model are left unchanged. The *lasso detection part* implements a counter for the number of transitions performed and adds the flag *live*. Finally, the specification is modified as described.

Generalization A more general form of the counter-based translation can use a flag *finished* instead of *looped*. That flag becomes true once a sufficient number of transitions has been performed to ensure that p would have occurred on a path if $\mathbf{F}p$ were true.

3.1.3 State-Recording Translation

Intuition In principle, state space search is memory-less. Detecting a loop as soon as it is closed can not be expressed directly in temporal logic. Instead, we add copies of all variables to the model, enabling us to save a state that has previously been visited. Reoccurrence of a state can now be detected by comparing the present state to the saved copy. As the start of a loop is not known beforehand, an oracle is used to indicate when a copy of the present state should be saved.

Example The counter-based and the state-recording translation differ only in the lasso detection part, see Fig. 3.3 (c). Here, it consists of an oracle *save*, a copy *hat_s* of the original state variables *s*, and an additional state variable, *lo* (for *l*ass*o*) to store the current position on the presumed lasso. lo has the value st for *st*em up to and including the point when *save* becomes true for the first time. The value of lo changes to lb (*l*oop *b*ody) once a state has been saved. It changes to lc (*l*oop *c*losed) after the second occurrence of the saved state has been detected. So far, the value of lo is only used to prevent overwriting the copy of the state variables.

Remark When the loop closing condition *looped* becomes true, the current state has been visited earlier. Therefore, the transformed specification does not need to take the current value of the property p into account. It suffices that the *live* flag remembers whether p has been true in the past. Figure 3.4 illustrates a run of the state-recording translation for the generic counterexample from Fig. 3.1.

---

*We assume that both formulae have to be true in all initial states.

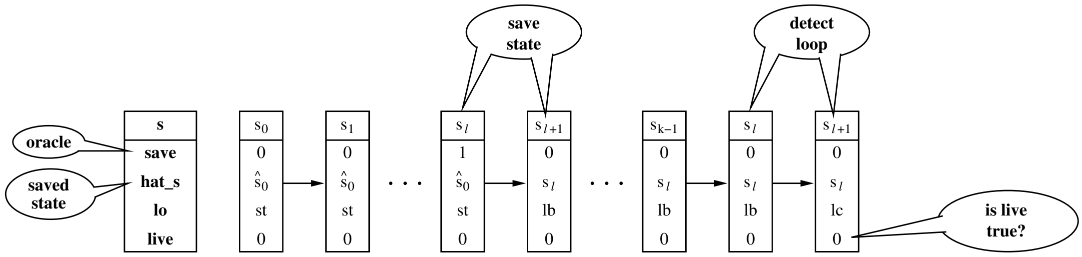


Figure 3.4: A run of the state-recording translation for the generic counterexample

3.1.4 Comparison

Finding bounds To work correctly, the counter-based translation requires coming up with a large enough bound. A trivial bound valid for arbitrary models is the overall number of states in the original model: any path of that length must include a loop. However, this requires an impractically large number of iterations in a realistic model as the property can only be checked when the counter has reached its bound. For most models and properties smaller bounds exist that still ensure correct results. A smaller bound adds fewer state bits and should lead to faster verification. Presently, a practically efficient method to compute a minimal bound is not known for arbitrary models and properties [CKOS05].

Shortest counterexamples Furthermore, the counter-based translation will, in general, not produce shortest counterexamples. Later in this chapter we show that the state-recording translation has this capability.

Generalization The counter-based translation gives no indication of a loop start. This makes generalization to arbitrary properties more difficult: the standard (automata-theoretic) approach to verify ω-regular properties using Büchi automata can not be applied directly, as it requires checking that a fair state has been seen on the loop.

One step ahead In our example of the counter-based translation in Fig. 3.3 (b), the last state in the loop must have already been seen and does not add new information regarding the truth of the liveness property. Therefore, the result could be determined one cycle before this bound is actually reached. This optimization has not been applied in Fig. 3.3 (b) to keep the presentation of both translations uniform. However, if an optimal bound were known for a model M and a property $\mathbf{F}p$, the counter-based translation could stop one step earlier than the state-recording translation if $M \models \mathbf{F}p$.

Focus As it does not need bounds and is easier to generalize, we concentrate on the state-recording translation below.

3.2 Translating Fair Repeated Reachability

3.2.1 First Attempt

Intuition Model checking of ω-regular properties can be done by detecting fair loops in the product of the model and a Büchi automaton for the negation of the property (see Sect. 2.8.1). A fairness condition is a set of states in the original system. A path is fair if it passes infinitely often through a state in each fairness condition. Recognizing fair loops in the state-recording translation is therefore similar to detecting that a simple liveness property has been fulfilled: an additional state variable f ($f$air) is introduced that observes similar to *live* whether one of its fair states has been seen. The invariant to check in the transformed system then becomes that a fair loop must never be closed.

Example Figure 3.5 shows an example. The counter is the same as in Fig. 3.3 but the specification now reads $\mathbf{F}\ \mathbf{G}\ s \neq 0$. The negation of this is $\mathbf{G}\ \mathbf{F}\ s = 0$, hence, a counterexample needs to see $s = 0$ infinitely often. We define the set of fair states to be $\{s = 0\}$. The part to save a state is unchanged. f has replaced *live*. It is initially set to false and becomes true when a fair state occurs *on the loop*, that is, when lo has the value lb. The transition of lo to lc now requires that f is true. I.e., $l = lc$ now signals detection of a fair loop.

3.2.2 Optimization

For Theory

Problems of the first attempt The translations shown in Figs. 3.3 (c) and 3.5 (b) recognize closure of a loop only with one step delay. In addition, they are forced to start on the stem. Hence, a lasso-shaped counterexample of length 1 will be signalled only at step 3. Figure 3.5 (c) shows an optimized version.

Optimized version The optimized state-recording translation is expressed as predicates on the initial states and the transition relation. The oracle *save* is not needed explicitly: it is replaced by the (now non-deterministic) transition of lo from st to lb. The set of initial states consists of two subsets. The first subset starts on the stem ($lo = st$). Correspondingly, f is false. The copy of the state variables of the original model are initialized with a default value. The second subset immediately starts the loop body ($lo = lb$), saves the initial state, and may set f to true if the initial state is fair. The transition relation is partitioned into subsets marked (1) – (5). Subset (1) covers the case on the stem. Fair states are irrelevant and the copies of their state variables keep their values. Subset (2) saves a state and enters the loop. Occurrence of a fair state may be remembered. Transitions from the third set (3) are taken as long as no second occurrence of the stored state or no fair state has been recorded. When f is true, a second occurrence is finally detected by a transition in (4). After that only transitions from the last set (5) are taken.

Remarks This version is the basis of the formalization of the translation. It detects presence of a fair loop at loop closure. Note also that neither a default initial value for the copies of the

```
MODULE main

VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1,s};
    s = 1: {2,s};
    s = 2: {3,s};
    s = 3: {0,s};
  esac;

LTLSPEC
  F G (s != 0)
```

(a) original

```
MODULE main

-- unmodified part of the
-- original system
VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1,s};
    s = 1: {2,s};
    s = 2: {3,s};
    s = 3: {0,s};
  esac;

-- fair lasso detection part
VAR
  save : boolean;
  hat_s: {0, 1, 2, 3};
  lo   : {st,lb,lc};
  f    : boolean;

ASSIGN
  init(lo) := st;
  next(lo) := case
    (lo = st) & save           : lb;
    (lo = lb) & (hat_s = s) & f: lc;
    1                          : lo;
  esac;

  init(hat_s) := 0;
  next(hat_s) := case
    (lo = st) & save: s;
    1               : hat_s;
  esac;

  init(f) := 0;
  next(f) := f | (lo = lb) & (s = 0);

-- transformed specification
INVARSPEC
  !(lo = lc)
```

(b) state-recording

```
MODULE main

-- unmodified part of the
-- original system
VAR
  s: {0, 1, 2, 3};
ASSIGN
  init(s) := 0;
  next(s) := case
    s = 0: {1,s};
    s = 1: {2,s};
    s = 2: {3,s};
    s = 3: {0,s};
  esac;

-- fair lasso detection part
VAR
  hat_s: {0, 1, 2, 3};
  lo   : {st,lb,lc};
  f    : boolean;

DEFINE
  hat_s_0 := 0;

INIT
    (lo = st & !f           & hat_s = hat_s_0)
  |
    (lo = lb & (f -> s = 0) & hat_s = s)

TRANS
  -- (1)
    (  lo = st & next(lo) = st
     & !f & !next(f)
     & hat_s = hat_s_0 & hat_s = next(hat_s))
  |

  -- (2)
    (  lo = st & next(lo) = lb
     & !f & (next(f) -> next(s) = 0)
     & hat_s = hat_s_0 & next(s) = next(hat_s))
  |

  -- (3)
    (  lo = lb & next(lo) = lb
     & (  (f -> next(f))
        & (next(f) -> f | next(s) = 0))
     & hat_s = next(hat_s))
  |

  -- (4)
    (  lo = lb & next(lo) = lc
     & f & next(f)
     & next(s) = hat_s & hat_s = next(hat_s))
  |

  -- (5)
    (  lo = lc & next(lo) = lc
     & f & next(f)
     & hat_s = next(hat_s))

-- transformed specification
INVARSPEC
  !(lo = lc)
```

(c) state-recording (optimized)

Figure 3.5: Translating fairness: NuSMV code of 2-bit counter with self-loops

Definition 1 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure with $\hat{s}_0 \in S$ arbitrary but fixed. Then $K^{\mathbf{S}} = (S^{\mathbf{S}}, T^{\mathbf{S}}, I^{\mathbf{S}}, L^{\mathbf{S}}, F^{\mathbf{S}})$ is defined as:*

$$
\begin{aligned}
S^{\mathbf{S}} &= S \times S \times \{st, lb, lc\} \times \mathbb{B} \\
I^{\mathbf{S}} &= \{(s_0, \hat{s}_0, st, 0) \mid s_0 \in I\} \cup \\
&\quad\ \{(s_0, s_0, lb, f) \mid s_0 \in I \wedge (f \rightarrow s_0 \in F_0)\} \\
T^{\mathbf{S}} &= \{((s, \hat{s}, lo, f), (s', \hat{s}', lo', f')) \mid (s, s') \in T \wedge \\
&\qquad ((lo = st \wedge lo' = st \wedge \neg f \wedge \neg f' \wedge \hat{s} = \hat{s}' = \hat{s}_0) \vee && (1) \\
&\qquad\ (lo = st \wedge lo' = lb \wedge \neg f \wedge (f' \rightarrow s' \in F_0) \wedge \hat{s} = \hat{s}_0 \wedge s' = \hat{s}') \vee && (2) \\
&\qquad\ (lo = lb \wedge lo' = lb \wedge (f \rightarrow f') \wedge (f' \rightarrow f \vee s' \in F_0) \wedge \hat{s} = \hat{s}') \vee && (3) \\
&\qquad\ (lo = lb \wedge lo' = lc \wedge f \wedge f' \wedge \hat{s} = s' = \hat{s}') \vee && (4) \\
&\qquad\ (lo = lc \wedge lo' = lc \wedge f \wedge f' \wedge \hat{s} = \hat{s}'))\} && (5) \\
L^{\mathbf{S}}((s, \hat{s}, lo, f)) &= L(s) \\
F^{\mathbf{S}} &= \emptyset
\end{aligned}
$$

state variables nor keeping the value of the copies constant in subsets (1) and (5) are necessary for correctness. Similarly, f need not be kept constant in subsets (1), (4), and (5). The advantage is improved complexity of the verification problem. Further, f is set nondeterministically in the initial state and in subsets (2) and (3) as this simplifies the calculation of the radius of the transformed model.

. . . and for Practice

While the representation of the transition relation in Fig. 3.5 (c) is well-suited for analysis (and, therefore, has been presented in that way here), in practice systems are more often formulated by (guarded) assignments of initial- and next-state values to state variables as in Fig. 3.5 (b). Hence, our actual implementation of the translation is based on that style but makes sure that counterexamples are detected when the loop is closed.

3.2.3 Formalization and Correctness

The formal definition of the state-recording translation of a Kripke structure K with a single fairness constraint is given in Def. 1. It largely corresponds to Fig. 3.5 (c). Theorem 5 states correctness: the language of (the original) K is non-empty iff a state with $lo = lc$ is reachable in (the transformed) $K^{\mathbf{S}}$.

Theorem 5 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure, let $K^{\mathbf{S}}$ be defined as above. Then*

$$Lang(K) \neq \emptyset \Leftrightarrow R(K^{\mathbf{S}}) \cap \{(s, \hat{s}, lc, f) \in S^{\mathbf{S}}\} \neq \emptyset$$

Proof: We prove the following bi-implications from top to bottom:

$$\begin{gathered}
\mathit{Lang}(K) \neq \emptyset \\
\Leftrightarrow \\
\exists \pi = (s_0 \dots s_{l-1})(s_l \dots s_{m-1} s_m \dots s_{k-1})^\omega \in \Pi^F \\
\text{with } k > m \geq l \geq 0 \wedge s_m \in F_0 \\
\Leftrightarrow \\
\exists \pi^{\mathbf{S}} = (s_0, \hat{s}_0, st, 0) \dots (s_{l-1}, \hat{s}_0, st, 0)(s_l, s_l, lb, 0) \dots (s_{m-1}, s_l, lb, 0)(s_m, s_l, lb, 1) \dots \\
\dots (s_{k-1}, s_l, lb, 1)(s_k, s_l, lc, 1) \in \Pi^{\mathbf{S}} \text{ with } k > m \geq l \geq 0 \\
\Leftrightarrow \\
R(K^{\mathbf{S}}) \cap \{(s, \hat{s}, lc, f) \in S^{\mathbf{S}}\} \neq \emptyset
\end{gathered}$$

1. "$\Rightarrow$": Every finite Kripke structure with non-empty language contains a lasso-shaped fair path, see also Sect. 2.8.2.

   "$\Leftarrow$": Obvious.

2. "$\Rightarrow$": Let $\pi = (s_0 \dots s_{l-1})(s_l \dots s_{m-1} s_m \dots s_{k-1})^\omega$ be an initialized fair path in K with $k > m \geq l \geq 0$ and $s_m \in F_0$. We construct $\pi^{\mathbf{S}}$ as follows.

   If $l > 0$ choose $\pi^{\mathbf{S}}[0] = (s_0, \hat{s}_0, st, 0)$ with arbitrary $\hat{s}_0$. Construct $(s_0, \hat{s}_0, st, 0) \dots (s_{l-1}, \hat{s}_0, st, 0)$ by taking transitions from subset (1). Assume first that $m > l$. Proceed to $(s_l, s_l, lb, 0)$ via a transition from (2), continue via $(s_{m-1}, s_l, lb, 0)$ to $(s_m, s_l, lb, 1)$ and from there to $(s_{k-1}, s_l, lb, 1)$ with $k - l - 1$ transitions from (3). As $k > l$ there is a transition from (4) to $(s_k, s_l, lc, 1)$ with $s_k = s_l$. If $m = l$, modify the target state of the transition from (2) to be $(s_l, s_l, lb, 1)$ and continue to $(s_{k-1}, s_l, lb, 1)$ and then to $(s_k, s_l, lc, 1)$, again with $s_k = s_l$.

   Otherwise, if $l = 0$, start with $(s_0, s_0, lb, 0)$ if $m > l$ and $(s_0, s_0, lb, 1)$ if $m = l$ and continue with $k - 1$ transitions from (3) and one from (4) as before.

   "$\Leftarrow$": Let $\pi^{\mathbf{S}} = (s_0, \hat{s}_0, st, 0) \dots (s_{l-1}, \hat{s}_0, st, 0)(s_l, s_l, lb, 0) \dots (s_{m-1}, s_l, lb, 0) \circ (s_m, s_l, lb, 1) \dots (s_{k-1}, s_l, lb, 1)(s_k, s_l, lc, 1)$ be an initialized path in $K^{\mathbf{S}}$ such that $k > m \geq l \geq 0$. From the construction of $K^{\mathbf{S}}$, $\pi' = s_0 \dots s_{l-1} s_l \dots s_{m-1} s_m \dots s_{k-1} s_k$ is an initialized finite path in K with $s_k = s_l$ and $s_m \in F_0$. Hence, $\pi = (s_0 \dots s_{l-1})(s_l \dots s_{m-1} s_m \dots s_{k-1})^\omega$ is an initialized fair path in K as desired.

3. "$\Rightarrow$": Obvious.

   "$\Leftarrow$": Let $s^{\mathbf{S}}$ be a reachable state in $\{(s, \hat{s}, lc, f) \in S^{\mathbf{S}}\}$, By definition of $K^{\mathbf{S}}$, f is 1. Further, there is an initialized path $\pi^{\mathbf{S}'}$ ending in $s^{\mathbf{S}}$. According to the definition of $T^{\mathbf{S}}$, $\pi^{\mathbf{S}'}$ takes precisely one transition from subset (4). Let $\pi^{\mathbf{S}}$ be the prefix of $\pi^{\mathbf{S}'}$ up to the target state of that transition. Let $k = |\pi^{\mathbf{S}}| - 1$. Clearly, $k > 0$. Let $\pi^{\mathbf{S}}[k] = (s_k, s_k, lc, 1)$. By definition of $T^{\mathbf{S}}$ there exists $0 \leq m' < k$ such that $\forall m' \leq i < k \,.\, \pi^{\mathbf{S}}[i] = (s_i, s_k, lb, 1)$ with $s_{m'} \in F_0$. Choose m to be the smallest such m'.

   Case 1 $m = 0$: With $l = 0$ and the definition of $I^{\mathbf{S}}$ and $T^{\mathbf{S}}$ we have that $\pi^{\mathbf{S}}[0] = (s_0, s_k, lb, 1) = (s_0, s_0, lb, 1)$.

   Case 2 $m > 0 \wedge \pi^{\mathbf{S}}[m-1] = (s_{m-1}, s_k, st, 0)$: Set $l = m$. By definition of $T^{\mathbf{S}}$, $\forall 0 \leq i < l \,.\, \pi^{\mathbf{S}}[i] = (s_i, \hat{s}_0, st, 0)$ and $\pi^{\mathbf{S}}[l] = (s_l, s_k, lb, 1) = (s_l, s_l, lb, 1)$.

Case 3 $m > 0 \wedge \pi^{\mathbf{S}}[m-1] = (s_{m-1}, s_k, lb, 0)$: By definition of $T^{\mathbf{S}}$ there is $0 \leq l' < m$ such that $\forall l' \leq i < m \;.\; \pi^{\mathbf{S}}[i] = (s_i, s_k, lb, 0)$. Set l to the smallest such l'.

Case 3.1 $l = 0$: By definition of $I^{\mathbf{S}}$, $\pi^{\mathbf{S}}[0] = (s_0, s_k, lb, 0) = (s_0, s_0, lb, 0)$.

Case 3.2 $l > 0$: From the definition of $T^{\mathbf{S}}$, $\forall 0 \leq i < l \;.\; \pi^{\mathbf{S}}[i] = (s_i, \hat{s}_0, st, 0)$ and $\pi^{\mathbf{S}}[l] = (s_l, s_k, lb, 0) = (s_l, s_l, lb, 0)$.

In all cases $s_k = s_l$ and the $\pi^{\mathbf{S}}$ has the desired shape.

$\square$

3.2.4 Extensions

Fairness A generalization to several fairness constraints can be achieved either by applying one of the translations in Sect. 2.3 or, more directly, by using one flag per fairness constraint. Extension to Muller, Rabin, or Streett acceptance conditions [Tho97] is also possible.

Hierarchy No special precautions are required for hierarchical models that can be flattened. If hierarchy should be preserved, the lo signal is defined[†] in the top-level module and only forwarded to each submodule. Each module defines the copy of its state variables and flags to remember occurrence of fair states. The non-determinism in the subsets (2) and (4) of Def. 1 ensures that corresponding transitions can be taken when all submodules are prepared to do so. This construction enables translating models (possibly by hand) without separate flattening before.

Example Figure 3.6 gives an example that includes hierarchy. Two tasks are trying to enter a critical section. If both are in their *try*-state a non-deterministic choice decides which task is allowed to proceed. Fairness ensures that each task eventually gets its turn. The example shows the translation of the mutex model with a specification given as a Büchi automaton. The original specification $\mathbf{G}\ ((\mathit{t0.s} = \mathit{try}) \rightarrow (\mathbf{F}\ (\mathit{t0.s} = \mathit{crit})))$ states that if task 0 is trying to enter its critical section, it will eventually be able to do so. The negated specification was translated into a Büchi automaton with Wring v1.1.0 (available from [Som]). The resulting automaton is depicted in Fig. 3.7.

3.3 Complexity

After correctness has been established, we can now state the theoretical bounds on the overhead for verification that is introduced into a system by our translation. Remember, that our objective was to enable checking ω-regular properties with techniques and tools previously only used for reachability calculation or safety checking. It turns out that the impact of our translations on the complexity for model checking or reachability calculation is quite reasonable. As sketched with the example of Fig. 3.5, the size of a non-canonical symbolic description in program code, increases only by a small constant factor.

[†] in the C sense [KR88]

```
MODULE task(id, turn)

-- unmodified part
VAR s: {non, try, crit};
ASSIGN
  init(s) := non;
  next(s) := case
    s = non: try;
    s = try & (id = turn): crit;
    s = try & !(id = turn): try;
    s = crit: non; esac;
FAIRNESS
  turn = id

MODULE main

VAR turn: 0..1;
    t0: task(0, turn);
    t1: task(1, turn);

-- Buechi automaton
VAR b: {n1, n2, n3, sink};
ASSIGN
  init(b) := {n2, n3};
  next(b) := case
    b = n1 & (t0.s = non | t0.s = crit): {n1};
    b = n1 & t0.s = try: {n2, n1};
    (b=n2 | b=n3) & (t0.s=non | t0.s=try): {n3};
    1: sink; esac;
FAIRNESS
  b = n3

-- Buechi specification
LTLSPEC F 0
```

(a) original

```
MODULE task(id, turn, lo)

-- unmodified part
VAR s: {non, try, crit};
ASSIGN
  init(s) := non;
  next(s) := case
    s = non: try;
    s = try & (id = turn): crit;
    s = try & !(id = turn): try;
    s = crit: non; esac;

-- fair lasso detection part
VAR hat_s: {non, try, crit};
    f    : boolean;
DEFINE
    hat_s_0 := non;
INIT
    (lo = st & !f               & hat_s = hat_s_0)
  | (lo = lb & (f -> turn = id) & hat_s = s)
TRANS
    (  lo = st & next(lo) = st
     & !f & !next(f)
     & hat_s = hat_s_0 & hat_s = next(hat_s))
  | (  lo = st & next(lo) = lb
     & !f & (next(f) -> next(turn) = id)
     & hat_s = hat_s_0 & next(s) = next(hat_s))
  | (  lo = lb & next(lo) = lb
     & (f -> next(f)) & ((next(f) -> f | next(turn) = id))
     & hat_s = next(hat_s))
  | (  lo = lb & next(lo) = lc
     & f & next(f)
     & next(s) = hat_s & hat_s = next(hat_s))
  | (  lo = lc & next(lo) = lc
     & f & next(f)
     & hat_s = next(hat_s))

MODULE main

VAR turn: 0..1;
    t0: task(0, turn, lo); -- note signal forwarding
    t1: task(1, turn, lo); -- note signal forwarding

-- Buechi automaton
VAR b: {n1, n2, n3, sink};
ASSIGN
  init(b) := {n2, n3};
  next(b) := case
    b = n1 & (t0.s = non | t0.s = crit): {n1};
    b = n1 & t0.s = try: {n2, n1};
    (b=n2 | b=n3) & (t0.s=non | t0.s=try): {n3};
    1: sink; esac;

-- fair lasso detection part
VAR hat_b: {n1, n2, n3, sink};
    lo   : {st,lb,lc};
    f    : boolean;
DEFINE
    hat_b_0 := n1;
INIT
    (lo = st & !f            & hat_b = hat_b_0)
  | (lo = lb & (f -> b = n3) & hat_b = b)
TRANS
    (  lo = st & next(lo) = st
     & !f & !next(f)
     & hat_b = hat_b_0 & hat_b = next(hat_b))
  | (  lo = st & next(lo) = lb
     & !f & (next(f) -> next(b) = n3)
     & hat_b = hat_b_0 & next(b) = next(hat_b))
  | (  lo = lb & next(lo) = lb
     & (f -> next(f)) & (next(f) -> f | next(b) = n3)
     & hat_b = next(hat_b))
  | (  lo = lb & next(lo) = lc
     & f & next(f)
     & next(b) = hat_b & hat_b = next(hat_b))
  | (  lo = lc & next(lo) = lc
     & f & next(f)
     & hat_b = next(hat_b))

-- transformed Buechi specification
INVARSPEC !(lo = lc)
```

(b) state-recording

Figure 3.6: Translating hierarchy: NuSMV code of mutex with Büchi specification

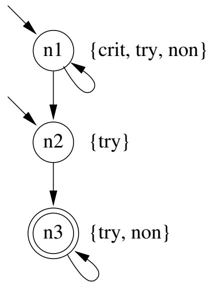


Figure 3.7: Büchi automaton for $\neg\mathbf{G}((s = \mathit{try}) \rightarrow (\mathbf{F}(s = \mathit{crit})))$

3.3.1 Explicit State Model Checking

Number of states In global (explicit) model checking [CE82] the complexity is governed by the number of states and the size of the transition relation. We first analyze the former, which increases quadratically:

$$|S^{\mathbf{S}}| = |S| \cdot |S| \cdot |\{st, lb, lc\}| \cdot |\{0,1\}| = 6 \cdot |S|^2 = O(|S|^2)$$

Number of reachable states In the case of on-the-fly (explicit) model checking [GPVW96] the size of the reachable state space $R(K^{\mathbf{S}})$ is of more interest. In a reachable state $(s, \hat{s}, lo, f) \in R(K^{\mathbf{S}})$, $\hat{s}$ is either the fixed initial state $\hat{s}_0$ or is reachable in K itself, since only reachable states are recorded. Therefore the size of $R(K^{\mathbf{S}})$ is bounded by

$$|R(K^{\mathbf{S}})| \leq |R(K)| \cdot |R(K)| \cdot |\{st, lb, lc\}| \cdot |\{0,1\}| = 6 \cdot |R(K)|^2 = \mathbf{O}(|R(K)|^2).$$

This bound is tight: a modulo n counter, like the model in Fig. 3.2 for $n = 4$, has $|R(K^{\mathbf{S}})| = \mathbf{O}(n^2)$ reachable states. If $n = 4$ then every combination of $\{0, \ldots, 3\} \times \{0, \ldots, 3\}$ can be reached for $(s, \hat{s})$.

Size of the transition relation For the size of the transition relation, note that Def. 1 fixes an initial state for the not-yet-saved copy of the original state variables and allows the value of the copy to change at most once on each path at the point of saving (subset (2)). This limits the size of $T^{\mathbf{S}}$ as follows: there are at most $|T|$ transitions in subsets (1) and (4), $2 \cdot |T|$ in subset (2), $2 \cdot |S| \cdot |T|$ in subset (3), and $|S| \cdot |T|$ in the last subset (5). Hence,

$$|T^{\mathbf{S}}| \leq 4 \cdot |T| + 3 \cdot |S| \cdot |T| = \mathbf{O}(|S| \cdot |T|)$$

Transitive closure of the transition relation The complexity of some model checking algorithms — e.g., pushdown systems, see Sect. 4.2 — depends also on the size of the transitive closure of the transition relation. For the transitive closure of $T^{\mathbf{S}}$, assume $((s, \hat{s}, lo, f), (s', \hat{s}', lo', f')) \in T^{\mathbf{S}*}$. Clearly, $(s, s') \in T^*$. There are 6 combinations of lo and lo' as shown in Tab. 3.1. Therefore we have

$$|T^{\mathbf{S}*}| \leq 9 \cdot |S| \cdot |T^*| + |T^*| = \mathbf{O}(|S| \cdot |T^*|)$$

lo	lo'	constraints		max. no. of transitions
st	st	$\hat{s} = \hat{s}' = \hat{s}_0,$	$\neg f \wedge \neg f'$	$\lvert T^*\rvert$
st	lb	$\hat{s} = \hat{s}_0,$	$\neg f$	$2 \cdot \lvert S\rvert \cdot \lvert T^*\rvert$
st	lc	$\hat{s} = \hat{s}_0,$	$\neg f \wedge f'$	$\lvert S\rvert \cdot \lvert T^*\rvert$
lb	lb	$\hat{s} = \hat{s}',$	$f \rightarrow f'$	$3 \cdot \lvert S\rvert \cdot \lvert T^*\rvert$
lb	lc	$\hat{s} = \hat{s}',$	f'	$2 \cdot \lvert S\rvert \cdot \lvert T^*\rvert$
lc	lc	$\hat{s} = \hat{s}',$	$f \wedge f'$	$\lvert S\rvert \cdot \lvert T^*\rvert$

Table 3.1: Deriving a bound on the number of transitions in the transitive closure

3.3.2 BDD-based Symbolic Model Checking

Static Bounds

BDD for transition relation Regarding symbolic model checking with BDDs [McM93] we have two results. First we relate the size of the BDDs for the transition relation of K and $K^{\mathbf{S}}$. Assuming S is encoded with $n = \lceil \log_2 |S| \rceil$ state bits, we can encode $S^{\mathbf{S}}$ with $2n + 3$ Boolean variables. It is important to interleave the Boolean variables for the original and copied instances of the state variables. Otherwise the size of the BDD for the term

$$\begin{array}{l} ((lo = st \wedge lo' = st \wedge \neg f \wedge \neg f' \wedge \hat{s} = \hat{s}' = \hat{s}_0) \vee) \\ \ (lo = st \wedge lo' = lb \wedge \neg f \wedge (f' \rightarrow s' \in F_0) \wedge \hat{s} = \hat{s}_0 \wedge s' = \hat{s}') \vee \\ \ (lo = lb \wedge lo' = lb \wedge (f \rightarrow f') \wedge (f' \rightarrow f \vee s' \in F_0) \wedge \hat{s} = \hat{s}') \vee \\ \ (lo = lb \wedge lo' = lc \wedge f \wedge f' \wedge \hat{s} = s' = \hat{s}') \vee \\ \ (lo = lc \wedge lo' = lc \wedge f \wedge f' \wedge \hat{s} = \hat{s}'))\} \end{array} \tag{3.2}$$

anded to the original transition relation T in Def. 1 explodes. With an interleaved order it is linear in n with a factor of approx. 20. The factor has been determined empirically for large state spaces as shown in Table 3.2. The first column shows the original number n of state bits. The second and third columns contain the number of BDD nodes necessary to represent Eqn. (3.2) using a non-interleaved (blocked) or interleaved order respectively. The exact number of nodes may vary with details of the encoding.

Assuming that a BDD representing the set of fair states has size c, the size of the BDD for $T^{\mathbf{S}}$ can be bounded roughly by $20 \cdot c \cdot n$ the size of the BDD for T by using the fact from [Bry86] that computing any Boolean binary operation on BDDs will produce a BDD that is linear in size with factor 1 in the size of the argument BDDs.

BDD for initial states Similar calculations for the set of initial states show that the size of BDDs representing $K^{\mathbf{S}}$ can be bound to be linear in the size of the BDDs representing K, linear in the number of state bits, and linear in the size of the BDD representing the set of fair states.

Dynamic Bounds

These *static* bounds do not say anything about the size of the BDDs in the fixed point iterations. The radius of a Kripke structure is an upper bound for the number of iterations necessary to reach a fixed point (see Sect. 2.8.3). Note that the results derived for the radius and the diameter

	blocked	interleaved	
n	nodes	nodes	nodes/n
10	9791	217	21.7
12	38985	257	21.4167
14	155731	297	21.2143
16	622685	337	21.0625
18	2490471	377	20.9444
20	$*$	417	20.85
32	$*$	657	20.5312
64	$*$	1297	20.2656
128	$*$	2577	20.1328
256	$*$	5137	20.0664
512	$*$	10257	20.0332
1024	$*$	20497	20.0166
2048	$*$	40977	20.0083
4096	$*$	81937	20.0042

Table 3.2: BDD sizes for Eqn. (3.2) ($*$ = memory limit of 512 MB reached).

of $K^{\mathbf{S}}$ stated in Theorem 4.4 of [BAS02] are incorrect if the $\neg p$-predicated diameter $d_{\neg p}$ [SB04] is larger than the diameter d. As shown in [SB04] the predicated diameter can be much larger than the diameter itself. A fixed analysis is given in [SB04]. Analysis of the construction in Def. 1 does not require predicated radius or diameter. This is the reason why the fairness flag is set non-deterministically.

Radius To determine the radius $r^{\mathbf{S}}$ of $K^{\mathbf{S}}$ consider an initial state $s_i^{\mathbf{S}} = (s_i, \hat{s}_i, lo_i, f_i)$ and a target state $s_t^{\mathbf{S}} = (s_t, \hat{s}_t, lo_t, f_t)$ with $s_i^{\mathbf{S}}, s_t^{\mathbf{S}} \in S \times S \times \{st, lb, lc\} \times \mathbb{B}$. If $s_t^{\mathbf{S}}$ is reachable from $s_i^{\mathbf{S}}$, $s_t^{\mathbf{S}}$ is reachable from $s_i^{\mathbf{S}}$ in at most $r^{\mathbf{S}}$ steps. This is denoted as follows:

$$s_i^{\mathbf{S}} = \begin{pmatrix} s_i \\ \hat{s}_i \\ lo_i \\ f_i \end{pmatrix} \xrightarrow{\leq r^{\mathbf{S}}} \begin{pmatrix} s_t \\ \hat{s}_t \\ lo_t \\ f_t \end{pmatrix} = s_t^{\mathbf{S}}$$

All enhancements to the original state space are monotonic in the added component. More specifically, Def. 1 fixes the following order of events: Starting from an initial state s_i, a loop state s_l must be saved. Only then can a fair state s_f be recorded. After that the loop state s_l may be reached to close the loop. A target state s_t may be reached following any of these intermediate states. More formally, 9 cases can be distinguished depending on lo_i, lo_t, f_i, f_t:

1. $lo_i = lb \wedge lo_t = lb \wedge f_i \wedge f_t$: the initial state is saved, its fairness recorded, but the loop

is not closed. In other words, this is simply a path from s_i to s_t.

$$\begin{pmatrix} s_i \\ s_i \\ lb \\ 1 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_t \\ s_i \\ lb \\ 1 \end{pmatrix}$$

2. $lo_i = lb \wedge lo_t = lc \wedge f_i \wedge f_t$: the initial state is saved, its fairness recorded, and the loop is closed. If s_i and s_t are not identical, the former must be reached a second time first, only then the path proceeds to s_t. As s_i is initial, the length of the second section (if present) is also bounded by r.

$$\begin{pmatrix} s_i \\ s_i \\ lb \\ 1 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_i \\ s_i \\ lc \\ 1 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_t \\ s_i \\ lc \\ 1 \end{pmatrix}$$

3. $lo_i = lb \wedge lo_t = lb \wedge \neg f_i \wedge \neg f_t$: the initial state is saved but neither is a fair state recorded nor is the loop closed.

$$\begin{pmatrix} s_i \\ s_i \\ lb \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_t \\ s_i \\ lb \\ 0 \end{pmatrix}$$

4. $lo_i = lb \wedge lo_t = lb \wedge \neg f_i \wedge f_t$: the initial state is saved, a fair state s_f is recorded (which might be identical to s_t), but the loop is not closed.

$$\begin{pmatrix} s_i \\ s_i \\ lb \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_f \\ s_i \\ lb \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_t \\ s_i \\ lb \\ 1 \end{pmatrix}$$

5. $lo_i = lb \wedge lo_t = lc \wedge \neg f_i \wedge f_t$: the initial state is saved, a fair state s_f is recorded, and the loop is closed. Note that s_f must be reached before before s_i is reached a second time. Again, the last section can be bounded by r. s_i and s_t might be the same states.

$$\begin{pmatrix} s_i \\ s_i \\ lb \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_f \\ s_i \\ lb \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_i \\ s_i \\ lc \\ 1 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_t \\ s_i \\ lc \\ 1 \end{pmatrix}$$

6. $lo_i = st \wedge lo_t = st \wedge \neg f_i \wedge \neg f_t$: no state is saved. Hence, simply a path from s_i to s_t.

$$\begin{pmatrix} s_i \\ \hat{s}_0 \\ st \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_t \\ \hat{s}_0 \\ st \\ 0 \end{pmatrix}$$

7. $lo_i = st \wedge lo_t = lb \wedge \neg f_i \wedge \neg f_t$: a state s_l is saved, but no fair state is recorded, and the loop is not closed. s_l and s_t could be identical.

$$\begin{pmatrix} s_i \\ \hat{s}_0 \\ st \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_l \\ \hat{s}_0 \\ lb \\ 0 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_t \\ \hat{s}_0 \\ lb \\ 0 \end{pmatrix}$$

8. $lo_i = st \wedge lo_t = lb \wedge \neg f_i \wedge f_t$: a state s_l is saved. Only after that can and is a fair state s_f reached. The loop is not closed. s_f and s_t can be the same.

$$\begin{pmatrix} s_i \\ \hat{s}_0 \\ st \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_l \\ \hat{s}_0 \\ lb \\ 0 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_f \\ \hat{s}_0 \\ lb \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_t \\ \hat{s}_0 \\ lb \\ 1 \end{pmatrix}$$

9. $lo_i = st \wedge lo_t = lc \wedge \neg f_i \wedge f_t$: A state s_l is saved, after that a fair state s_f is reached, and only then the loop is closed. s_l and s_t could be identical states.

$$\begin{pmatrix} s_i \\ \hat{s}_0 \\ st \\ 0 \end{pmatrix} \xrightarrow{\leq r} \begin{pmatrix} s_l \\ \hat{s}_0 \\ lb \\ 0 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_f \\ \hat{s}_0 \\ lb \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_l \\ \hat{s}_0 \\ lc \\ 1 \end{pmatrix} \xrightarrow{\leq d} \begin{pmatrix} s_t \\ \hat{s}_0 \\ lc \\ 1 \end{pmatrix}$$

Diameter Bounds on the diameter $d^{\mathbf{S}}$ can be obtained similarly by starting in an arbitrary state $s_s^{\mathbf{S}} = (s_s, \hat{s}_s, lo_s, f_s)$. As we have defined the length of a path as its number of states, we obtain

$$r^{\mathbf{S}} \leq r + 3 \cdot d - 3 = \mathbf{O}(d)$$

and

$$d^{\mathbf{S}} \leq 4 \cdot d - 3 = \mathbf{O}(d)$$

Note though, that the diameter of a system can be exponentially larger than its radius as shown in [SB04].

3.3.3 Summary

Theorem 6 summarizes some of the results of this section. It supports the intuition that $K^{\mathbf{S}}$ corresponds to $|S|$ copies of K running in parallel, each with a different guess of the loop start.

Theorem 6 *Let $K = (S, T, I, L, F = \{F_0\})$ be a Kripke structure with $r = r(K)$ and $d = d(K)$. Let $K^{\mathbf{S}}$ be defined as in Def. 1. Then*

1. $|S^{\mathbf{S}}| = |S| \cdot |S| \cdot |\{st, lb, lc\}| \cdot |\{0, 1\}| = 6 \cdot |S|^2 = O(|S|^2)$
2. $|R(K^{\mathbf{S}})| \leq 6 \cdot |R(K)|^2 = \mathbf{O}(|R(K)|^2)$
3. $|T^{\mathbf{S}}| \leq 4 \cdot |T| + 3 \cdot |S| \cdot |T| = \mathbf{O}(|S| \cdot |T|)$
4. $|T^{\mathbf{S}*}| \leq 9 \cdot |S| \cdot |T^*| + |T^*| = \mathbf{O}(|S| \cdot |T^*|)$
5. $r^{\mathbf{S}} \leq r + 3 \cdot d - 3 = \mathbf{O}(d)$
6. $d^{\mathbf{S}} \leq 4 \cdot d - 3 = \mathbf{O}(d)$

□

3.4 Shortest Counterexamples

When performing reachability analysis on $K^{\mathbf{S}}$, the algorithm will either reach a fixed point or find a counterexample after at most $r^{\mathbf{S}}+1$ iterations, from which a lasso-shaped counterexample in K can be derived. Moreover, if the property under consideration is false and if breadth-first search is used for reachability analysis in $K^{\mathbf{S}}$, the proof of Thm. 5 implies that a shortest lasso-shaped counterexample in K (i.e., with respect to the product of the automaton for the property and that for the original model to be verified) can be derived. If the approach using several flags to encode general fairness is used (see Sect. 2.3) and the property is encoded using a tight Büchi automaton (see Sect. 5.1), this implies that the counterexample is a shortest one with respect to the property in the original model to be verified. Note, that the translated system needs one step to detect a loop. Hence, when lasso-shaped counterexamples and violating prefixes (see Sect. 2.7) are searched for in parallel, and search is stopped after finding the first counterexample, a reported shortest violating prefix may be one state longer than the length of a potential shortest lasso-shaped counterexample.

3.5 Related Work

3.5.1 Reduction to and Power of Reachability Checking

Reduction to reachability Burch [Bur91] presents an idea how to verify liveness properties as safety properties by using timed trace structures [Bur89, Dil88]. Both, model and specification are given as timed trace structures. Discrete time is modeled using time ticks. The user is then required to provide a mapping which translates time ticks of the model to those of the specification. Burch claims that his method is conservative, i.e., an invalid mapping may only lead to false positives. He reports that a CTL model checker was faster than his translation on an example. He concludes that his method "may not be efficient in practice" and is of interest mainly in theory or if no dedicated liveness checking is available. While the exposition [Bur91] is by example, we believe that his approach resembles the counter-based translation 3.1.2 if no further knowledge of the timing behavior of the model is available. We suspect that applying his method to an arbitrary (untimed) finite state system in order to verify a simple liveness property $\mathbf{F}p$ requires adding a time tick to each cycle or dead-end state[‡] in the model and then checking that the number of time ticks before a p-state is reached does not exceed the maximal number of time ticks on any initialized loop-free $\neg p$-path.

Shilov et al. have developed a game-theoretic reduction from their Second Order Elementary Propositional Dynamic Logic (SOEPDL) [SY01] to reachability for classes of models which include all finite models and which are closed under Cartesian product and power set [SYE+05, SY01]. SOEPDL is more expressive than Stirling's second order propositional modal logic 2M [Sti96] (i.e., it subsumes LTL, CTL, and the propositional μ-calculus [Koz83]). While the reduction by Shilov et al. is more powerful than our reduction, in terms of number of configurations, [SYE+05] is doubly exponential where ours is typically quadratic. In the words of [SYE+05], this renders it "totally non-efficient, impractical". They also introduce the notion of *equal model checking power* of logics with different expressive power for a particular class of models: two logics LG, LG' have equal model checking power for a class of models MD if,

[‡]Timed trace theory [Bur89] assumes finite traces.

for every $M \in MD, \phi \in LG$, the model checking problem $M \models \phi$ can be reduced to $M' \models \phi'$ where $M' \in MD$, $\phi' \in LG'$, and M' is obtained from M by simple algebraic transformation such as Cartesian product and power set construction.

Ultes-Nitsche [UN02] shows that, for any ω-regular property ϕ and model M, satisfaction of ϕ in M *within fairness* [NW97] corresponds to classical satisfaction as defined in Sect. 2.3 of some ϕ' in M where ϕ' is a safety property depending on ϕ and M. Satisfaction within fairness disregards a potential counterexample π in M if every finite prefix $\pi[0..i]$ can be continued to a witness of ϕ in M. In other words, the semantics [NW97] disregards a potential counterexample on which M continuously chooses not to fulfil ϕ although it always could. By adding some state bits and strong transition fairness (e.g., [Pel01]) one can construct an implementation M' with the same set of finite behaviors as M that fulfills ϕ [NW97]. As a consequence, to detect that a model satisfies ϕ within fairness but not in the classical sense the model needs to be observed infinitely long. As this is practically impossible, Ultes-Nitsche concludes that classical satisfaction seems too fine-grained [UN02]. However, it is not clear that satisfaction within fairness is always the desired semantics. Too see this, consider the following fairness constraint that might be necessary to guarantee classical satisfaction:

> The device, which the programmer forgot to ask for whether it's present and which the program now may be waiting for indefinitely, will finally be plugged in by the user.

Our reduction preserves the classical semantics.

Loop detection As mentioned in Sect. 1.3, the basic idea of using loop detection to find accepting paths is taken from explicit on-the-fly and SAT-based symbolic bounded model checking [CVWY92, BCCZ99]. Sistla and Clarke also use guessing of a loop start in their proof of PSPACE completeness of the model checking problem for PLTLB [SC85]. While they, too, save the state at the loop start, they guess the length of the loop and check consistency between the last state of the loop and the saved first state of the loop when the end of the loop is reached.

Restriction to reachability Jard and Jéron propose on-the-fly model checking for a subset of PLTLF using reachability [JJ90]. Beer et al. extend CTL with regular expressions and syntactically characterize a subset of formulae that can be checked with forward reachability [BBDL98]. They obtain significant time and space savings. In their experience, more than 80% of all practical formulae belong to that subset. Kupferman and Vardi's work [KV01] shares the idea to translate a safety property into a finite state automaton on finite words, which enables forward reachability, but provides more complete results from a theoretical point of view. For a discussion of [KV01] see Sect. 2.7.

Other Related in spirit is the well-known programming technique to detect presence of a cycle in a linked list: Initialize two pointers to point to the first and second elements, respectively. Move them forward through the list where the second pointer moves at twice the speed of the first pointer. Stop with report of a cycle when they point to the same element, terminate without reporting a cycle when the faster pointer reaches the end of the list.[§] Finally, iterative

[§]Thanks to Irina Tuduce for the reminder.

squaring [BCM+92] non-deterministically guesses intermediate states of a path to speed up computation of the transitive closure of a (transition) relation; similarly, but deterministically, Savitch [Sav70] tries candidate intermediate states of computation sequences in his reduction from non-deterministic space to deterministic space complexities of Turing machines.

3.5.2 Shortest Counterexamples

BDD-based symbolic and explicit-state model checking Finding a shortest counterexample for a general property amounts to finding a shortest fair cycle, which is an NP-complete problem [CGMZ95]. Most BDD-based model checkers offer only heuristics to minimize the length of counterexamples to such properties. For a comparative study on their performance and the length of the generated counterexamples see [RBS00]. The double DFS [CVWY92] typically used to search the state space in explicit state model checking does not find shortest counterexamples. Gastin et al. propose an algorithm to minimize the length of counterexamples, which may visit a state an exponential number of times [GMZ04]. Hansen and Kervinen [HK05] suggest a polynomial time, linear space algorithm. While experimental results show more regular behavior than [GMZ04], the algorithm uses BFS and backwards exploration of the state space, making it somewhat unpractical. A lazy algorithm is used by Latvala and Heljanko to find short counterexamples in Streett automata [LH00].

Bounded model checking The first technique in widespread use that can produce shortest counterexamples for general LTL properties is SAT-based bounded model checking [BCCZ99]. While [BCCZ99] was restricted to future time LTL, more recent implementations cover full LTL [BC03], [CRS04], [LBHJ05]. Whether shortest counterexamples can be reported depends also on the encoding of the property. Both, [BC03] and [LBHJ05] find shortest counterexamples. [CRS04] achieves higher performance than [BC03] but sacrifices shortest counterexamples. A detailed experimental comparison of [CRS04] and [LBHJ05] is not yet available.

Easy-to-read counterexamples The shortest counterexample is not necessarily the easiest one to understand. Jin, Ravi, and Somenzi annotate those parts of a counterexample that constitute inevitable progress to the error [JRS02]. Ravi and Somenzi then continue by removing irrelevant events [RS04]. “Nice” (usually small) rather than arbitrary values of variables can also make a counterexample easier to read. Groce and Kroening provide a solution for SAT-based model checking, where that problem is most relevant [GK05].

Explaining counterexamples Recently, automated approaches to explain a counterexample and locate the error have been proposed. Groce’s thesis on the topic contains many references [Gro05].

Other [CV03] is a survey on counterexamples and [Gro05] also gives more general references.

3.6 Summary

We have presented a source-to-source translation from fair repeated reachability to reachability by adding a copy of the state variables to the original system, non-deterministically saving a current state, and detecting a second occurrence of the saved state after all fairness constraints have been met. The translation leads to an increase by a factor of $|S|$ (where S is the set of states in the original system) for most parameters relevant to explicit state model checking. Radius and diameter, which are more relevant for symbolic model checking, grow only by a small constant factor. If forward breadth-first reachability analysis is used the translation helps to find shortest lasso-shaped counterexamples.

4

Extending to Infinite State Systems

We all know Linux is great ... it does infinite loops in 5 seconds.
Linus Torvalds

In this section we extend the state-recording translation to some classes of infinite state systems, which have received considerable attention in the past and for which verification tools are available: ($\omega-$)regular model checking [KMM$^+$01, WB98, BJNT00, BLW04a, AJNd03], pushdown systems [BEM97, FWW97, EHRS00a, ES01], and timed automata [AD94, LPY97].

For each of these classes specialized sets of notations have developed. To aid a reader, who is familiar with some of these classes, we adopt the notation used in some of the major publications on each class of systems. Therefore, we present required notation for each class at the beginning of the section describing its reduction rather than in Chap. 2.

We do not include fairness constraints in loop detection to simplify the exposition. Fairness can be handled in the same way as for finite state systems.

4.1 Regular Model Checking

4.1.1 Preliminaries

The notation in this section is mostly borrowed from [BJNT00]. Let Σ be a finite alphabet. Regular sets (respectively relations) can be represented as finite-state automata (resp. transducers). These are given as four tuple (Q, q_0, δ, F) where Q is a finite set of states, q_0 is the initial state, $\delta : (Q \times \Sigma) \mapsto 2^Q$ (resp. $\delta : (Q \times (\Sigma \times \Sigma)) \mapsto 2^Q$) is the transition function, and $F \subseteq Q$* is the set of accepting states.

A relation $R \subseteq \Sigma^* \times \Sigma^*$ is *length-preserving* iff $\forall (w, w') \in R \,.\, |w| = |w'|$. A *program* is a triple $\mathcal{P} = (\Sigma, \Phi_I, R)$ where $\Phi_I \subseteq \Sigma^*$ is a regular set of *initial configurations* and $R \subseteq \Sigma^* \times \Sigma^*$ is a regular, length-preserving *transition relation*.

A *configuration* of a program $\mathcal{P}$ is a word w over Σ. *Paths* are finite or infinite sequences of configurations $\pi = \pi[0]\pi[1]\ldots$, such that $\forall 0 < i < |\pi| \,.\, (\pi[i-1], \pi[i]) \in R$. A path is *initialized* if $\pi[0] \in \Phi_I$. $\Pi(\mathcal{P})$ is the set of paths of $\mathcal{P}$.

*We do not have to deal with fairness constraints in this chapter, hence, there is no ambiguity.

Definition 2 *Let $\mathcal{P} = (\Sigma, \Phi_I, R)$ be a program with $\hat{a}_0 \in \Sigma$ arbitrary but fixed. Then $\mathcal{P}^{\mathbf{S}} = (\Sigma^{\mathbf{S}}, {\Phi_I}^{\mathbf{S}}, R^{\mathbf{S}})$ is defined as*

$$
\begin{aligned}
\Sigma^{\mathbf{S}} &= \{st, lb, lc\} \cup (\Sigma \times \Sigma) \\
{\Phi_I}^{\mathbf{S}} &= st \circ \{w \times \hat{w} \in (\Sigma \times \Sigma)^* \mid |w| = |\hat{w}| \wedge w \in \Phi_I \wedge \hat{w} = {\hat{a}_0}^*\} \cup \\
&\quad\; lb \circ \{w \times w \in (\Sigma \times \Sigma)^* \mid w \in \Phi_I\} \\
R^{\mathbf{S}} &= \{((lo \circ (w \times \hat{w})), (lo' \circ (w' \times \hat{w}'))) \subseteq (\{st, lb, lc\} \circ (\Sigma \times \Sigma)^*)^2 \mid \\
&\quad\; |w| = |\hat{w}| = |w'| = |\hat{w}'| \wedge (w, w') \in R \wedge \\
&\quad\; ((lo = st \wedge lo' = st \wedge \hat{w} = \hat{w}' = {\hat{a}_0}^*) \vee && (1) \\
&\quad\;\; (lo = st \wedge lo' = lb \wedge \hat{w} = {\hat{a}_0}^* \wedge w' = \hat{w}') \vee && (2) \\
&\quad\;\; (lo = lb \wedge lo' = lb \wedge \hat{w} = \hat{w}') \vee && (3) \\
&\quad\;\; (lo = lb \wedge lo' = lc \wedge \hat{w} = w' = \hat{w}') \vee && (4) \\
&\quad\;\; (lo = lc \wedge lo' = lc \wedge \hat{w} = \hat{w}'))\} && (5)
\end{aligned}
$$

4.1.2 Reduction

Intuition and formal definition In the finite case the state to be saved was simply added as a separate component to the state of the transformed system. A finite automaton can only remember a finite amount of information. Hence, in order to apply the reduction to regular model checking it is not possible to construct an automaton that first reads a state of the original program and compares that with a saved copy. Instead, we extend the alphabet of the program to tuples of letters to store and compare states position by position of a word. Other than that, the construction in Def. 2 is the same as in the finite case.[†]

Still a program The following Lemma 7 shows that the reduced program is still a program.

Lemma 7 *If $\mathcal{P} = (\Sigma, \Phi_I, R)$ is a program, so is $\mathcal{P}^{\mathbf{S}} = (\Sigma^{\mathbf{S}}, {\Phi_I}^{\mathbf{S}}, R^{\mathbf{S}})$.*

Proof: Assume that Φ_I is given by $(Q_I, {q_0}_I, \delta_I, F_I)$. To represent an automaton (not) saving the initial state we use separate copies of $(Q_I, {q_0}_I, \delta_I, F_I)$, $(Q_I^{\neq}, {q_0}_I^{\neq}, \delta_I^{\neq}, F_I^{\neq})$ and $(Q_I^{=}, {q_0}_I^{=}, \delta_I^{=}, f_I^{=})$. Then $({Q_I}^{\mathbf{S}}, {{q_0}_I}^{\mathbf{S}}, {\delta_I}^{\mathbf{S}}, {F_I}^{\mathbf{S}})$ with

$$
\begin{aligned}
{Q_I}^{\mathbf{S}} &= Q_I^{\neq} \cup Q_I^{=} \cup \{q_{lo}\}, \\
{{q_0}_I}^{\mathbf{S}} &= q_{lo}, \\
{\delta_I}^{\mathbf{S}} &= \{(q_{lo}, st, {q_0}_I^{\neq})\} \cup \{(q^{\neq}, (a, \hat{a}_0), q^{\neq\prime}) \mid (q^{\neq}, a, q^{\neq\prime}) \in \delta_I^{\neq}\} \cup \\
&\quad\; \{(q_{lo}, lb, {q_0}_I^{=})\} \cup \{(q^{=}, (a, a), q^{=\prime}) \mid (q^{=}, a, q^{=\prime}) \in \delta_I^{=}\}, \text{ and} \\
{F_I}^{\mathbf{S}} &= F_I^{\neq} \cup F_I^{=},
\end{aligned}
$$

is a finite automaton accepting ${\Phi_I}^{\mathbf{S}}$.

Similarly, if R is given by $(Q_R, {q_0}_R, \delta_R, F_R)$, we construct a finite transducer $({Q_R}^{\mathbf{S}}, {{q_0}_R}^{\mathbf{S}}, {\delta_R}^{\mathbf{S}}, {F_R}^{\mathbf{S}})$ to accept $R^{\mathbf{S}}$. We use separate copies of $(Q_R, {q_0}_R, \delta_R, F_R)$ to leave the saved word unchanged and check for it being ${\hat{a}_0}^*$ (superscript ^{1}, corresponding to disjunct 1 in Def. 2), save a word (sup. ^{2}, corr. to subset (2)), leave the saved word unchanged (sup. ^{35},

[†]Remember that the cross product of sequences is defined component-wise, i.e., it returns a sequence of tuples rather than a tuple of sequences.

corr. to subsets (3) and (5)), and compare current and stored word (sup. ^{4}, corr. to subset (4)).

$$
\begin{array}{lcl}
{Q_R}^{\mathbf{S}} & = & Q^1_R \cup Q^2_R \cup Q^{35}_R \cup Q^4_R \cup \{q_{lo}\}, \\
{q_0}_R^{\mathbf{S}} & = & q_{lo}, \\
{\delta_R}^{\mathbf{S}} & = & \{(q_{lo}, (st, st), q^1_0), (q_{lo}, (st, lb), q^2_0), (q_{lo}, (lb, lb), q^{35}_0), (q_{lo}, (lb, lc), q^4_0), \\
& & \quad (q_{lo}, (lc, lc), q^{35}_0)\} \cup \\
& & \{(q^1, ((a, \hat{a}_0), (a', \hat{a}_0)), q^{1\prime}) \mid (q^1, (a, a'), q^{1\prime}) \in \delta^1_R\} \cup \\
& & \{(q^2, ((a, \hat{a}_0), (a', a')), q^{2\prime}) \mid (q^2, (a, a'), q^{2\prime}) \in \delta^2_R\} \cup \\
& & \{(q^{35}, ((a, \hat{a}), (a', \hat{a})), q^{35\prime}) \mid (q^{35}, (a, a'), q^{35\prime}) \in \delta^{35}_R\} \cup \\
& & \{(q^4, ((a, a'), (a', a')), q^{4\prime}) \mid (q^4, (a, a'), q^{4\prime}) \in \delta^4_R\}, \text{ and} \\
{F_R}^{\mathbf{S}} & = & F^1_R \cup F^2_R \cup F^{35}_R \cup F^4_R
\end{array}
$$

□

Correctness Theorem 8 establishes correctness of the reduction.

Theorem 8 *Let $\mathcal{P} = (\Sigma, \Phi_I, R)$ be a program, $\mathcal{P}^{\mathbf{S}}$ be defined as above, and $\hat{w}_I \in {\hat{a}_0}^*$ with $|\hat{w}_I| = |w_0|$. Assume $k > l \geq 0$.*

$$(w_0 \dots w_{l-1})(w_l \dots w_{k-1})^\omega \in \Pi(\mathcal{P})$$
$$\Leftrightarrow$$
$$(st \circ (w_0 \times \hat{w}_I)) \dots (st \circ (w_{l-1} \times \hat{w}_I))(lb \circ (w_l \times w_l)) \dots (lb \circ (w_{k-1} \times w_l))(lc \circ (w_k \times w_l)) \in \Pi(\mathcal{P}^{\mathbf{S}})$$

Proof: Analogous to the proof of Thm. 5. □

Complexity Note, that each transition of ${\delta_I}^{\mathbf{S}}$ is marked with a pair of letters whose second element is either $\hat{a}_0$ or identical to the first element. Similar observations can be made for transitions between states with superscripts 2 and 4 in ${\delta_R}^{\mathbf{S}}$. Hence, we trivially have

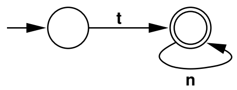
Theorem 9 *Let $\mathcal{P} = (\Sigma, \Phi_I, R)$ be a program, and $\mathcal{P}^{\mathbf{S}}$ be defined as above. Then*

$$
\begin{array}{lclclcl}
|{Q_I}^{\mathbf{S}}| & = & 2|Q_I| + 1 & \qquad & |{\delta_I}^{\mathbf{S}}| & = & 2 \cdot |\delta_I| + 2 \\
|{Q_R}^{\mathbf{S}}| & = & 4|Q_R| + 1 & & |{\delta_R}^{\mathbf{S}}| & = & (|\Sigma| + 3) \cdot |\delta_R| + 5
\end{array}
$$

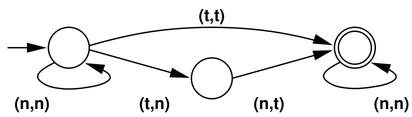
4.1.3 Example

Token passing and original system As an example of a parameterized system, consider token passing as used, e.g., in [BJNT00]. An array of processes passes a single token from left to right. Initially, the leftmost process has the token. Each transition either leaves the token where it is, or passes it on to the right neighbor of the current owner. Processes can be in states t or n depending on whether they do (t) or don't have (n) the token. Hence, $\Sigma = \{n, t\}$. An automaton and a transducer representing the initial states Φ_I and the transition relation R are shown in Fig. 4.1 (a) and (b).

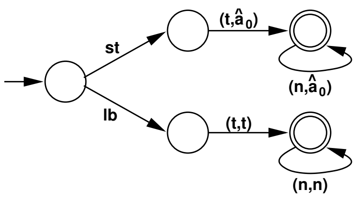
The transformed system According to Def. 2, $\Sigma^{\mathbf{S}} = \{st, lb, lc\} \cup \{(n, n), (n, t), (t, n), (t, t)\}$. ${\Phi_I}^{\mathbf{S}}$ and $R^{\mathbf{S}}$ are given in Fig. 4.1 (c) and (d). From top to bottom, the two (four) main branches of the automaton (transducer) correspond to the state sets $Q^{\neq}_I$ and $Q^{=}_I$ (Q^1_R, Q^2_R, Q^{35}_R, and Q^4_R), respectively.



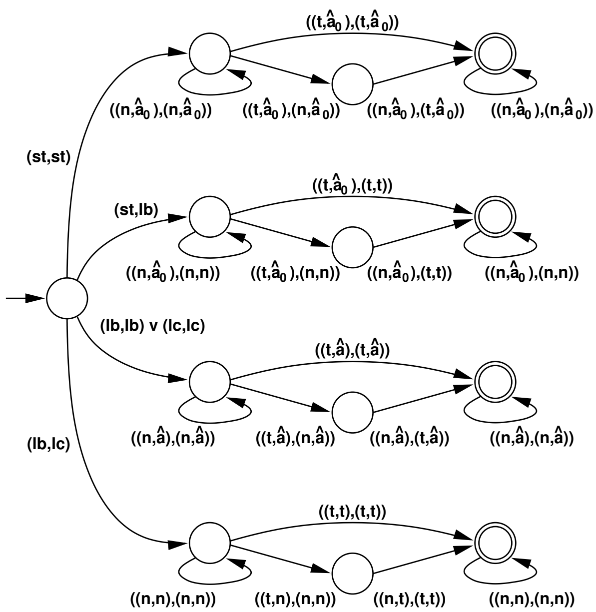
(a) initial configurations Φ_I



(b) transition relation R



(c) reduced initial configurations $\Phi_I{}^{\mathbf{S}}$



(d) reduced transition relation $R^{\mathbf{S}}$

Figure 4.1: Example: token passing [BJNT00]

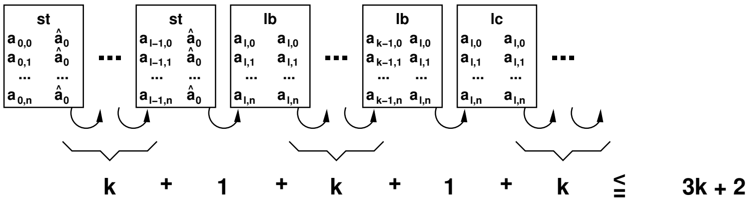


Figure 4.2: The reduction preserves boundedness of local depth.

4.1.4 Discussion

On termination Checking reachability for a program $\mathcal{P} = (\Sigma, \Phi_I, R)$ is undecidable in general [AK86]. Not only does this exclude a sound and complete algorithm for regular model checking, but it also raises the question whether $\mathcal{P}^{\mathsf{S}}$ can be verified by a given algorithm if $\mathcal{P}$ can. We have the following partial result: Bouajjani et al. developed a technique to compute the transitive closure of a regular relation R [BJNT00, JN00]. A sufficient criterion for termination of that computation is *bounded local depth* [BJNT00, JN00] of R. Our construction preserves that property. Intuitively, a relation has local depth k if for any $(w, w') \in R^+$ each position in w needs to be rewritten no more than k times. Note that in any path π^{S} of $\mathcal{P}^{\mathsf{S}}$ the projection of π^{S} onto lo will be a prefix of $st^*\ lb^+\ lc^+$. Furthermore, $\hat{w}$ changes its value in π^{S} at most once at the transition of lo from st to lb. Hence, with similar reasoning as for radius and diameter in Sect. 3.3 we can infer that, if R has local depth k, R^{S} has local depth $\leq 3k + 2$. The factor of 3 increases if fairness constraints are added. For an illustration see Fig. 4.2.

Shortest Counterexamples As is, the transitive closure construction of [BJNT00, JN00] does not preserve sufficient information to find a shortest counterexample. One could therefore determine truth or falsity of a given specification using the transitive closure [BJNT00, JN00] to reach a fixed point also in the case of an infinite radius. If the specification turns out to be false, standard reachability checking (i.e., without acceleration) can be used to determine a shortest counterexample, which has necessarily finite distance from the set of initial configurations. A counterexample with shortest configurations (representing, e.g., the smallest number of processes in a parameterized system) can be easily obtained once the reachable set of "bad" configurations has been computed: choose the shortest bad configuration and search for a (shortest) path to that configuration using finite state model checking.

ω-regular model checking The ideas of regular model checking have been extended to infinite words by regarding the finite automata used to represent sets of states and the transition relation as Büchi automata on infinite words [BLW04a]. The techniques of [BLW04a] require the Büchi automata to be *weakly deterministic*. A Büchi automaton is weak (1) if each of its strongly connected components contains either only accepting or only non-accepting states and (2) if the set of states can be partitioned into an ordered set of subsets such that each path in the automaton progresses in descending order through these subsets. From the proof of Lemma 7 it's easy to see that, if B is a weakly deterministic Büchi automaton (for the set of initial configurations) or transducer (for the transition relation), so is B^{S}. Clearly, repeated reachability may not be sufficient to verify general LTL properties for $\omega-$regular programs.

A program $\mathcal{P} = (\Sigma, \Phi_I, R)$ is an example of a system that satisfies the following property:

> If there is a counterexample in P to an ω-regular property ϕ then there is also a counterexample in P to ϕ with a finite number of different configurations.

If this property is satisfied for some model M and ω-regular property ϕ, application of the state-recording translation is sound with respect to the decision problem $M \overset{?}{\models} \phi$, i.e., a "bad" configuration is reachable in M^{S} if ϕ does not hold in M. While the systems in the next two sections do not satisfy that property, existence of an infinite fair path can still always be deduced by storing and comparing a finite amount of information.

4.2 Pushdown Systems

4.2.1 Preliminaries

Notation in this section is along the lines of [EHRS00a]. A *pushdown system* M is a four tuple $M = (P, \Gamma, \Delta, C_I)$ where P is a finite set of *control locations*, Γ is a finite *stack alphabet*, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *transition rules*, and $C_I \subseteq P \times \Gamma$ is a finite set of *initial configurations*.

A *configuration* is a pair $\langle p, w \rangle$[‡] with $p \in P$ and $w \in \Gamma^*$. A *path* is a (finite or infinite) sequence of configurations $\pi = \pi[0]\pi[1]\ldots$, where $\pi[i] = \langle p_i, w_i \rangle$, such that $\forall i < |\pi| - 1 \ .\ \exists \gamma_i \in \Gamma, \exists u_i, v_i \in \Gamma^* \ .\ w_i = \gamma_i v_i \wedge w_{i+1} = u_i v_i \wedge ((p_i, \gamma_i), (p_{i+1}, u_i)) \in \Delta$. A path is *initialized* if $\pi[0] \in C_I$. $\Pi(M)$ is the set of paths of M.

A *head* is a pair $\langle p, \gamma \rangle$ with $p \in P$ and $\gamma \in \Gamma$. If $c = \langle p, \gamma w \rangle$ is a configuration, $head(c) = \langle p, \gamma \rangle$. A head $\langle p, \gamma \rangle$ is *repeating* if there exist a path π in M and $w \in \Gamma^*$ such that $|\pi| > 1$, $\pi[0] = \langle p, \gamma \rangle$, and $\pi[|\pi| - 1] = \langle p, \gamma w \rangle$. $heads(\pi)$ denotes the sequence of heads derived from a path π.

Bouajjani et al. proved [BEM97] that (1) every path that ends in a configuration with a repeating head can be extended to an infinite path, and (2) from every infinite path π a path $\sigma\tau$ can be derived such that $|\sigma| < \infty$ and $heads(\tau) = (\langle p_0, \gamma_0 \rangle \ldots \langle p_{l-1}, \gamma_{l-1} \rangle)^\omega$. I.e., if there exists an infinite path in M, then there also exists one whose sequence of heads forms a lasso.

4.2.2 Reduction

Intuition Based on the results of [BEM97] it is sufficient to find repeating heads when checking PLTLB formulae on pushdown systems. Hence, a reduction of repeated reachability to reachability need only store and watch out for a second occurrence of a repeating head $\langle p, \gamma \rangle$ rather than an entire configuration. However, to infer from the second occurrence of a head that this head is indeed repeating, one has to ensure that the stack height between the first and the second occurrence never fell below the stack height at the first occurrence. To this end the stack alphabet is extended such that each stack symbol has an additional flag bs (*b*ottom of *s*tack) to remember a given stack height. When saving a head, this flag is set for the bottom element pushed on the stack in the post-configuration. Whenever an element with $bs = 1$ is removed from the stack without being replaced in the same transition, a *l*oop *e*rror flag le is set.

[‡] Note that we do not need the notation for lassos in this chapter.

A minor difference to previous reductions In the previous examples, $lo = lc$ signals a second occurrence of a configuration immediately at that occurrence. However, the definition of the transition rules for pushdown systems may not give access to the topmost element of the stack in the post-configuration. If no new element is pushed on the stack a comparison with a stored stack element cannot be performed. For this reason we introduce a one-state delay in the case of pushdown systems for lo and the stored head. Hence, there is no need for an initial configuration with that configuration already saved.

Formal definition Definition 3 shows the entire reduction. The transition relation is partitioned into 5 sets again. While no state has been saved (subset (1)), $lo = st$ and $\neg le$ remain constant, the initial values for $\hat{p}$ and $\hat{\gamma}$ are just copied, and no stack height need be remembered (bs_0 is false). Saving a state (subset (2)) can only occur if a non-empty word is pushed back on the stack — otherwise, the next transition would immediately violate the above-mentioned condition for the stack height of a repeating head. Taking a transition from subset (2) saves the head $\langle p, \gamma \rangle$ (in the pre-configuration) in $\hat{p}$ and $\hat{\gamma}$ (in the post-configuration), sets lo to lb, and marks the current stack height by setting bs to true for the bottom element pushed on the stack. Transitions from subset (3) are taken while a second occurrence of the stored head has not been seen, hence, lo as well as $\hat{p}$ and $\hat{\gamma}$ keep their values. In addition, the condition not to fall below the stack height at the time of saving is checked. When this is the case, i.e., when an element with bs true is popped from the stack and only an empty word is pushed back, the loop error flag le is set to true. This prevents signalling a repeating head in the future by restricting subsequent transitions to subset (3). When the stack height remains above the required level, le keeps its value and the flag bs is set in the bottom element of the word pushed onto the stack iff it was set in the symbol popped from the stack. A second occurrence of $\langle p, \gamma \rangle$ is signalled by setting $lo = lc$ when taking a transition from subset (4). le, $\hat{p}$, and $\hat{\gamma}$ keep their values. Any remembered stack height is discarded. Transitions of the last subset (5) keep all additional location components constant.

Correctness In the following we prove correctness of the reduction.

Theorem 10 *Let $M = (P, \Gamma, \Delta, c_I)$ be a pushdown system and $M^{\mathbf{S}}$ be defined as above. There exists an initialized path π to a repeating head $\langle p_0, \gamma \rangle$ in M if and only if there exists an initialized path $\pi^{\mathbf{S}}$ in $M^{\mathbf{S}}$ with $\pi^{\mathbf{S}}[|\pi^{\mathbf{S}}| - 2] = \langle (p_0, p_0, \gamma, lb, 0), w_{|\pi^{\mathbf{S}}|-2} \rangle$, where $w_{|\pi^{\mathbf{S}}|-2}(0) = \gamma$, and $\pi^{\mathbf{S}}[|\pi^{\mathbf{S}}| - 1] = \langle (p, p_0, \gamma, lc, 0), w_{|\pi^{\mathbf{S}}|-1} \rangle$.*

Proof: "$\Rightarrow$": Assume an initialized path π to a repeatable head $\langle p_0, \gamma \rangle$. Hence, there exist $l \geq 0$, $q_0, \ldots, q_{l-1} \in P$, $w_0, \ldots, w_{l-1} \in \Gamma^*$, $v \in \Gamma^*$ where $\forall i < l \,.\, \pi[i] = \langle q_i, w_i \rangle$ and $\pi[l] = \langle p_0, \gamma v \rangle$.

By the definition of a repeating head there are $k > l$, $p_1, \ldots, p_{k-l-1} \in P$, $u_0, \ldots, u_{k-l} \in \Gamma^+$, where $u_0 = u_{k-l}[0] = \gamma$, such that π can be extended to an infinite path $\pi^{inf} \in \Pi(M)$:

$$\begin{aligned} \forall i < l \,.\, \pi^{inf}[i] &= \pi[i] \\ \forall i \geq l \,.\, \pi^{inf}[i] &= \langle p_{(i-l) \bmod (k-l)}, \\ &\quad u_{(i-l) \bmod (k-l)} (u_{k-l}[1] \ldots u_{k-l}[|u_{k-l}| - 1])^{(i-l) \operatorname{div} (k-l)} v \rangle \end{aligned}$$

Definition 3 *Let $M = (P, \Gamma, \Delta, C_I)$ be a pushdown system, let $(\hat{p}_I, \hat{\gamma}_I) \in P \times \Gamma$ be arbitrary but fixed. Then, $M^{\mathbf{S}} = (P^{\mathbf{S}}, \Gamma^{\mathbf{S}}, \Delta^{\mathbf{S}}, {C_I}^{\mathbf{S}})$ is defined as*

$$
\begin{aligned}
P^{\mathbf{S}} &= P \times P \times \Gamma \times \{st, lb, lc\} \times \mathbb{B} \\
\Gamma^{\mathbf{S}} &= \Gamma \times \mathbb{B} \\
\Delta^{\mathbf{S}} &= \{(((p, \hat{p}, \hat{\gamma}, lo, le), (\gamma, bs)), ((p', \hat{p}', \hat{\gamma}', lo', le'), (w'[0], bs'_h) \ldots (w'[|w'|-1], bs'_0))) \mid \\
&\quad (((p, \gamma), (p', w')) \in \Delta) \wedge \\
&\quad (|w'| > 1 \rightarrow \neg bs'_h \wedge \ldots \wedge \neg bs'_1) \wedge \\
&\quad ((lo = st \wedge lo' = st \wedge \neg le \wedge \neg le' \wedge \hat{p} = \hat{p}' = \hat{p}_I \wedge \hat{\gamma} = \hat{\gamma}' = \hat{\gamma}_I \wedge \\
&\quad\quad (|w'| > 0 \rightarrow \neg bs'_0)) \vee & (1) \\
&\quad (lo = st \wedge lo' = lb \wedge \neg le \wedge \neg le' \wedge p = \hat{p}' \wedge \hat{p} = \hat{p}_I \wedge \gamma = \hat{\gamma}' \wedge \hat{\gamma} = \hat{\gamma}_I \wedge \\
&\quad\quad (|w'| > 0) \wedge bs'_0) \vee & (2) \\
&\quad (lo = lb \wedge lo' = lb \wedge ((|w'| = 0 \wedge bs \vee le) \leftrightarrow le') \wedge \\
&\quad\quad \hat{p} = \hat{p}' \wedge \hat{\gamma} = \hat{\gamma}' \wedge (|w'| > 0 \rightarrow (bs \leftrightarrow bs'_0))) \vee & (3) \\
&\quad (lo = lb \wedge lo' = lc \wedge \neg le \wedge \neg le' \wedge p = \hat{p} = \hat{p}' \wedge \gamma = \hat{\gamma} = \hat{\gamma}' \wedge \\
&\quad\quad (|w'| > 0 \rightarrow \neg bs'_0)) \vee & (4) \\
&\quad (lo = lc \wedge lo' = lc \wedge \neg le \wedge \neg le' \wedge \hat{p} = \hat{p}' \wedge \hat{\gamma} = \hat{\gamma}' \wedge (|w'| > 0 \rightarrow \neg bs'_0)))\} & (5) \\
{C_I}^{\mathbf{S}} &= \{\langle (p_I, \hat{p}_I, \hat{\gamma}_I, st, 0), (\gamma_I, 0) \rangle \mid \langle p_I, \gamma_I \rangle \in C_I\}
\end{aligned}
$$

From that we construct an initialized finite path $\pi^{\mathbf{S}}$ as follows:

$$
\begin{aligned}
&\forall i < l \;.\; \pi^{\mathbf{S}}[i] = \langle (q_i, \hat{p}_I, \hat{\gamma}_I, st, 0), w_i \times 0^{|w_i|} \rangle \\
&\pi^{\mathbf{S}}[l] = \langle (p_0, \hat{p}_I, \hat{\gamma}_I, st, 0), (\gamma, 0) \circ (v \times 0^{|v|}) \rangle \\
&\pi^{\mathbf{S}}[l+1] = \langle (p_1, p_0, \gamma, lb, 0), (u_1 \times 0^{|u_1|-1} 1) \circ (v \times 0^{|v|}) \rangle \\
&\forall l+1 < i < l+k \;.\; \pi^{\mathbf{S}}[i] = \langle (p_{i-l}, p_0, \gamma, lb, 0), (u_{i-l} \times 0^{|u_{i-l}|-1} 1) \circ (v \times 0^{|v|}) \rangle
\end{aligned}
$$

if $|u_{k-l}| > 1$

$$
\begin{aligned}
\pi^{\mathbf{S}}[k] = \langle (p_0, p_0, \gamma, lb, 0), (\gamma, 0) \circ \\
\circ ((u_{k-l}[1], 0) \ldots (u_{k-l}[|u_{k-l}| - 2], 0)(u_{k-l}[|u_{k-l}| - 1], 1)) \circ (v \times 0^{|v|}) \rangle \\
\pi^{\mathbf{S}}[k+1] = \langle (p_1, p_0, \gamma, lc, 0), (u_1 \times 0^{|u_1|}) \circ \\
\circ ((u_{k-l}(1), 0) \ldots (u_{k-l}[|u_{k-l}| - 2], 0)(u_{k-l}[|u_{k-l}| - 1], 1)) \circ (v \times 0^{|v|}) \rangle
\end{aligned}
$$

otherwise

$$
\begin{aligned}
\pi^{\mathbf{S}}[k] &= \langle (p_0, p_0, \gamma, lb, 0), (\gamma, 1) \circ (v \times 0^{|v|}) \rangle \\
\pi^{\mathbf{S}}[k+1] &= \langle (p_1, p_0, \gamma, lc, 0), (u_1 \times 0^{|u_1|}) \circ (v \times 0^{|v|}) \rangle
\end{aligned}
$$

"$\Leftarrow$": Assume an initialized path $\pi^{\mathbf{S}}$ to $\pi^{\mathbf{S}}[|\pi^{\mathbf{S}}| - 2] = \langle (p_0, p_0, \gamma, lb, 0), w_{|\pi^{\mathbf{S}}|-2} \rangle$, where $w_{|\pi^{\mathbf{S}}|-2}[0] = \gamma$, and $\pi^{\mathbf{S}}[|\pi^{\mathbf{S}}| - 1] = \langle (p_1, p_0, \gamma, lc, 0), w_{|\pi^{\mathbf{S}}|-1} \rangle$. By Def. 3, $\exists 0 < l < |\pi^{\mathbf{S}}| - 2$ such that $\pi^{\mathbf{S}}[l] = \langle (p_0, \hat{p}_I, \hat{\gamma}_I, st, 0), w_l \times 0^{|w_l|} \rangle$ and $w_l[0] = \gamma$. Clearly, the projection of $\pi^{\mathbf{S}}[0, l]$ on the first components of its state and stack is an initialized path in M to a repeatable head. $\square$

4.2.3 Complexity

Locations and transitions The following theorem states the number of locations and transitions in the transformed system.

Theorem 11 *Let $M = (P, \Gamma, \Delta, C_I)$ be a pushdown system. $M^{\mathbf{S}}$ has $\mathbf{O}(|P||\Gamma||P|)$ locations and $\mathbf{O}(|P||\Gamma||\Delta|)$ transition rules.*

Proof: The locations of M are extended in $M^{\mathbf{S}}$ to store another location, a stack symbol, and a small constant amount of additional state information. For $\Delta^{\mathbf{S}}$, there are $\mathbf{O}(|\Delta|)$ transition rules in subsets (1), (2), and (4), and $\mathbf{O}(|P||\Gamma||\Delta|)$ in (3) and (5). □

Selecting an algorithm for analysis A number of algorithms has been proposed that can be used to check reachability in a pushdown system (e.g., [BEM97, FWW97, EHRS00a]). [EHRS00a] improves on previous results, the algorithms (for forward and for backward reachability) as well as their analysis are clearly formulated, and an implementation [ES01] is available. We therefore chose [EHRS00a] as the basis for a more detailed complexity analysis of our reduction. Below we give full details for the (more complicated) case of forward reachability and only state the result for backward reachability, which can be obtained in a very similar way.

Analyzing forward reachability Algorithm 3 in [EHRS00a] can be used to check reachability for a pushdown system $M = (P, \Gamma, \Delta, C_I)$ where $(p, \gamma, p', w') \in \Delta \Rightarrow |w'| \leq 2$. The algorithm takes a finite state automaton $A_M = (\Gamma, Q, \delta, P, F)$, which accepts a set of configurations of M, as input. The stack alphabet Γ is the input alphabet of A_M. The set of states Q consists of the locations of M, P, and internal states, Q_1. P is also the set of initial states, states in $F \subseteq Q$ are final. δ is the transition relation. The algorithm transforms A_M into $A'_M = (\Gamma, Q', \mathit{rel}, P, F')$, which accepts the configurations that are reachable from configurations accepted by A_M. The state set Q is extended with a set Q_2. It contains one state q_r for each transition rule $r \in \Delta$ such that $|w'| = 2$. If A_M is an automaton that accepts the set of initial configurations C_I, δ has size $\mathbf{O}(|P||\Gamma|)$ and Q_1 only requires a single final state q_f. In this case, the set of reachable configurations can be computed in $\mathbf{O}(|P||\Delta|^2 + |P||\Gamma|)$ time.

In the following we show that a blow-up of $\mathbf{O}(|P||\Gamma|)$ is sufficient when the algorithm is applied to $M^{\mathbf{S}} = (P^{\mathbf{S}}, \Gamma^{\mathbf{S}}, \Delta^{\mathbf{S}}, {C_I}^{\mathbf{S}})$. Note that $M^{\mathbf{S}}$ has a single, fixed initial value for each of the added location components, and saving of current location and top stack symbol may only occur once in each path of $M^{\mathbf{S}}$. Intuitively, as in the finite case, this amounts to checking $|P||\Gamma|$ versions of M in parallel, rather than a system with $\mathbf{O}(|P||\Gamma||P|)$ locations and $\mathbf{O}(|P||\Gamma||\Delta|)$ transition rules. The next lemma establishes that on any sequence of states of $A_{M^{\mathbf{S}}}$ the stored location and stack symbol, which are present in all states of $A_{M^{\mathbf{S}}}$ other than q_f, exhibit at most one change from some $(\hat{p}, \hat{\gamma})$ to the initial values $(\hat{p}_I, \hat{\gamma}_I)$. Theorem 13 then proves the overall result.

Below we identify a state $q_{r^{\mathbf{S}}}$ with a transition rule $r^{\mathbf{S}}$. Therefore, we use $p(p^{\mathbf{S}}), p(q), \hat{p}(p^{\mathbf{S}}), \hat{p}(q), \ldots$ to refer to the components of a state $p^{\mathbf{S}} \in P^{\mathbf{S}}$ or $q \in P^{\mathbf{S}} \cup Q_2$, and also $p'(q), \hat{p}'(q), \ldots$ if $q \in Q_2$.

Lemma 12 *Let $M = (P, \Gamma, \Delta, C_I)$ be a pushdown system where $(p, \gamma, p', w') \in \Delta \Rightarrow |w'| \leq 2$. When applied to $M^{\mathbf{S}}$ with $A_{M^{\mathbf{S}}}$ accepting ${C_I}^{\mathbf{S}}$, Algorithm 3 in [EHRS00a] adds only transitions $p^{\mathbf{S}} \xrightarrow{\gamma^{\mathbf{S}}} q'$ to trans and $q \xrightarrow{\gamma^{\mathbf{S}}} q'$ to rel with $p^{\mathbf{S}} \in P^{\mathbf{S}}$, $q \in P^{\mathbf{S}} \cup Q_2$, $q' \in \{q_f\} \cup Q_2$, and*

$$\begin{array}{lcl} q' \in Q_2 & \Rightarrow & \hat{p}(p^{\mathbf{S}}) = \hat{p}'(q') \wedge \hat{\gamma}(p^{\mathbf{S}}) = \hat{\gamma}'(q') \vee \hat{p}'(q') = \hat{p}_I \wedge \hat{\gamma}'(q') = \hat{\gamma}_I \\ q \in P^{\mathbf{S}} \wedge q' \in Q_2 & \Rightarrow & \hat{p}(q) = \hat{p}'(q') \wedge \hat{\gamma}(q) = \hat{\gamma}'(q') \vee \hat{p}'(q') = \hat{p}_I \wedge \hat{\gamma}'(q') = \hat{\gamma}_I \\ q, q' \in Q_2 & \Rightarrow & \hat{p}'(q) = \hat{p}'(q') \wedge \hat{\gamma}'(q) = \hat{\gamma}'(q') \vee \hat{p}'(q') = \hat{p}_I \wedge \hat{\gamma}'(q') = \hat{\gamma}_I \end{array}$$

Further, it only adds states $p^{\mathbf{S}}$ to $eps(q)$ if $q \in \{q_f\} \cup Q_2$ and such that

$$p^{\mathbf{S}} \in eps(q) \wedge q \in Q_2 \quad \Rightarrow \quad \hat{p}(p^{\mathbf{S}}) = \hat{p}'(q) \wedge \hat{\gamma}(p^{\mathbf{S}}) = \hat{\gamma}'(q) \vee \hat{p}'(q) = \hat{p}_I \wedge \hat{\gamma}'(q) = \hat{\gamma}_I$$

Proof: (by induction)

Base case, lines 1 – 6. For each $({p_I}^{\mathbf{S}}, {\gamma_I}^{\mathbf{S}}) \in {C_I}^{\mathbf{S}}$, line 1 adds ${p_I}^{\mathbf{S}} \xrightarrow{{\gamma_I}^{\mathbf{S}}} q_f$, ${p_I}^{\mathbf{S}} \in P^{\mathbf{S}}$ to $trans$. rel is initialized to $\emptyset$ in line 2. For each $r^{\mathbf{S}} = (p^{\mathbf{S}}, \gamma^{\mathbf{S}}, p^{\mathbf{S}\prime}, \gamma_1^{\mathbf{S}\prime}\gamma_0^{\mathbf{S}\prime}) \in \Delta^{\mathbf{S}}$ line 5 adds $p^{\mathbf{S}\prime} \xrightarrow{\gamma_1^{\mathbf{S}\prime}} q_{r^{\mathbf{S}}}$ to $trans$, where $p^{\mathbf{S}\prime} \in P^{\mathbf{S}}$ and $q_{r^{\mathbf{S}}} \in Q_2$. Line 6 sets $eps(q)$ to $\emptyset$ for all q.

Inductive case, lines 7 – 22. By i.a. the claim holds for line 10. For lines 11 – 22, note that $\Delta^{\mathbf{S}}$ has only transitions $r^{\mathbf{S}}$ with $\hat{p}(r^{\mathbf{S}}) = \hat{p}'(r^{\mathbf{S}}) \wedge \hat{\gamma}(r^{\mathbf{S}}) = \hat{\gamma}'(r^{\mathbf{S}}) \vee \hat{p}(r^{\mathbf{S}}) = \hat{p}_I \wedge \hat{\gamma}(r^{\mathbf{S}}) = \hat{\gamma}_I$. Let $p^{\mathbf{S}} \xrightarrow{\gamma^{\mathbf{S}}} q \in trans$ and $r^{\mathbf{S}} \in \Delta^{\mathbf{S}}$ such that $p^{\mathbf{S}} = p^{\mathbf{S}}(r^{\mathbf{S}}) \wedge \gamma^{\mathbf{S}} = \gamma^{\mathbf{S}}(r^{\mathbf{S}})$.

Case 1 $|w^{\mathbf{S}\prime}(r^{\mathbf{S}})| = 1$: Line 18 adds $p^{\mathbf{S}\prime}(r^{\mathbf{S}}) \xrightarrow{\gamma_0^{\mathbf{S}\prime}(r^{\mathbf{S}})} q$ to $trans$. Clearly, $p^{\mathbf{S}\prime}(r^{\mathbf{S}}) \in P^{\mathbf{S}}$.

- **Case 1.1** $lo(p^{\mathbf{S}}) = st$: By construction of $\Delta^{\mathbf{S}}$, $\hat{p}(p^{\mathbf{S}}) = \hat{p}_I$ and $\hat{\gamma}(r^{\mathbf{S}}) = \hat{\gamma}(p^{\mathbf{S}}) = \hat{\gamma}_I$. By i.a., $q \in \{q_f\} \cup Q_2$ and $q \in Q_2 \Rightarrow \hat{p}'(q) = \hat{p}_I \wedge \hat{\gamma}'(q) = \hat{\gamma}_I$.
- **Case 1.2** $lo(p^{\mathbf{S}}) \neq st$: By assumption and construction of $\Delta^{\mathbf{S}}$, $\hat{p}'(r^{\mathbf{S}}) = \hat{p}(r^{\mathbf{S}}) = \hat{p}(p^{\mathbf{S}})$ and $\hat{\gamma}'(r^{\mathbf{S}}) = \hat{\gamma}(r^{\mathbf{S}}) = \hat{\gamma}(p^{\mathbf{S}})$. By i.a., $q \in \{q_f\} \cup Q_2$ and $q \in Q_2 \Rightarrow \hat{p}'(q) = \hat{p}'(r^{\mathbf{S}}) \wedge \hat{\gamma}'(q) = \hat{\gamma}'(r^{\mathbf{S}}) \vee \hat{p}'(q) = \hat{p}_I \wedge \hat{\gamma}'(q) = \hat{\gamma}_I$.

Case 2 $|w^{\mathbf{S}\prime}(r^{\mathbf{S}})| = 2$: Line 20 adds $q_{r^{\mathbf{S}}} \xrightarrow{\gamma_0^{\mathbf{S}\prime}(r^{\mathbf{S}})} q$ to rel. $q_{r^{\mathbf{S}}} \in Q_2$, the rest of the proof is analogous to Case 1. Line 22 adds $p^{\mathbf{S}\prime\prime} \xrightarrow{\gamma_0^{\mathbf{S}\prime}(r^{\mathbf{S}})} q$ to $trans$ for each $p^{\mathbf{S}\prime\prime} \in eps(q_{r^{\mathbf{S}}})$. The claim follows by i.a. on $p^{\mathbf{S}\prime\prime} \in eps(q_{r^{\mathbf{S}}})$ and $q_{r^{\mathbf{S}}} \xrightarrow{\gamma_0^{\mathbf{S}\prime}(r^{\mathbf{S}})} q \in rel$.

Case 3 $|w^{\mathbf{S}\prime}(r^{\mathbf{S}})| = 0$: Line 13 adds $p^{\mathbf{S}\prime}(r^{\mathbf{S}})$ to $eps(q)$, where $p^{\mathbf{S}\prime}(r^{\mathbf{S}}) \in P^{\mathbf{S}}$. The proof of the claim for $eps(q)$ is analogous to Case 1. Let $q \xrightarrow{\gamma^{\mathbf{S}\prime}} q' \in rel$. Line 15 adds $p^{\mathbf{S}\prime}(r^{\mathbf{S}}) \xrightarrow{\gamma^{\mathbf{S}\prime}} q'$ to $trans$. The claim follows by i.a. on $q \xrightarrow{\gamma^{\mathbf{S}\prime}} q' \in rel$.

□

Theorem 13 *Let $M = (P, \Gamma, \Delta, C_I)$ be a pushdown system such that $(p, \gamma, p', w') \in \Delta \Rightarrow |w'| \leq 2$. Algorithm 3 in [EHRS00a] runs on $M^{\mathbf{S}}$, with $A_{M^{\mathbf{S}}}$ accepting ${C_I}^{\mathbf{S}}$, in time and space*

$$\mathbf{O}(|P||\Gamma|(|P||\Delta|^2))$$

Proof: The proof is with the previous Lemma along the lines of the original proof in [EHRS00b]. We therefore only give the time complexity for each of the lines as used in the proof [EHRS00b] in Tab. 4.1. □

line	time $M^{\mathbf{S}}$	remark
init	$\mathbf{O}(\|P\|\|\Gamma\|(\|\Delta\|))$	Sort each $(p^{\mathbf{S}}, \gamma^{\mathbf{S}}, p^{\mathbf{S}\prime}, w^{\mathbf{S}\prime}) \in \Delta^{\mathbf{S}}$ into buckets according to $(p^{\mathbf{S}}, \gamma^{\mathbf{S}})$. With Thm. 11 there are $\mathbf{O}(\|P\|\|\Gamma\|\|\Delta\|)$ transition rules.
1	$\mathbf{O}(\|P\|\|\Gamma\|)$	Each added location component has a single fixed initial value.
2	$\mathbf{O}(\|P\|\|\Gamma\|(\|P\|))$	$rel = \emptyset$, $Q = P^{\mathbf{S}} \cup \{q_f\}$, and $F = \{q_f\}$.
5	$\mathbf{O}(\|P\|\|\Gamma\|(\|\Delta\|))$	See *init*.
6	$\mathbf{O}(\|P\|\|\Gamma\|(\|P\|\|\Delta\|))$	According to Lemma 12 we need to store at most $\mathbf{O}(\|P\|)$ states for each of the $\mathbf{O}(\|P\|\|\Gamma\|\|\Delta\|)$ states in Q_2.
8	$\mathbf{O}(\|P\|\|\Gamma\|(\|P\|\|\Delta\|^2) + \|P\|\|\Gamma\|)$	Executed at most once for every transition in δ or added in lines 15, 18, and 22.
15	$\mathbf{O}(\|P\|\|\Gamma\|(\|P\|\|\Delta\|^2))$	Q_1 only consists of q_f with no outgoing transitions. Further, we have $\mathbf{O}(\|P\|\|\Gamma\|\|\Delta\|)$ states q in Q_2. By Lemma 12 the number of target states q' is limited to $\mathbf{O}(\|\Delta\|)$, the number of source states p' is $\mathbf{O}(\|P\|)$.
18	$\mathbf{O}(\|P\|\|\Gamma\|(\|\Delta\|^2))$	There are $\mathbf{O}(\|P\|\|\Gamma\|\|\Delta\|)$ transitions in $\Delta^{\mathbf{S}}$. By Lemma 12 we have $\mathbf{O}(\|\Delta\|)$ combinations of a particular $p^{\mathbf{S}\prime}$ and q'.
20	$\mathbf{O}(\|P\|\|\Gamma\|(\|\Delta\|^2))$	Like line 18.
22	$\mathbf{O}(\|P\|\|\Gamma\|(\|P\|\|\Delta\|^2))$	By Lemma 12 there are at most $\mathbf{O}(\|P\|)$ states in each $eps(q_{r}\mathbf{s})$.

Table 4.1: Time complexity for algorithm 3 in [EHRS00a] when applied to pushdown system $M^{\mathbf{S}}$

						γ_2	γ_2											γ_2	γ_2					
					γ_1	γ_1	γ_1	γ_1									γ_1	γ_1	γ_1	γ_1				
w(i)				γ_0	γ_0	γ_0	γ_0	γ_0	γ_0							γ_0	γ_0	γ_0	γ_0	γ_0	γ_0			
			γ_2	γ_2	γ_2	γ_2	γ_2	γ_2	γ_2	γ_2					γ_2	γ_2	γ_2	γ_2	γ_2	γ_2	γ_2	γ_2		
		γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1			γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	γ_1	
	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0	γ_0
$p(i)$	p_0	p_0	p_0	p_1	p_1	p_1	p_2	p_2	p_2	p_3	p_3	p_3	p_4	p_4	p_4	p_5	p_5	p_5	p_2	p_2	p_2	p_3	p_3	p_3
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Figure 4.3: The soonest second occurrence of a repeating head might not indicate the shortest counterexample.

Analyzing backward reachability If the algorithm for backward reachability is used instead, the increase in complexity is similar. A finite state automaton that accepts all "bad" configurations $\langle(p,\hat{p},\hat{\gamma},lc,0),w\rangle$ has $\mathbf{O}(|P|^2|\Gamma|)$ states and $\mathbf{O}(|P|^2|\Gamma|^2)$ transitions. With similar reasoning as above we obtain:

Theorem 14 *Let $M = (P,\Gamma,\Delta,C_I)$ be a pushdown system such that $(p,\gamma,p',w') \in \Delta \Rightarrow |w'| \leq 2$. Algorithm 1 in [EHRS00a] computes the set of configurations from which a configuration in $\{\langle(p,\hat{p},\hat{\gamma},lc,0),w\rangle \mid p,\hat{p} \in P \wedge \hat{\gamma} \in \Gamma \wedge w \in (\Sigma,\mathbb{B})^*\}$ is reachable in $M^{\mathbf{S}}$ in time*

$$\mathbf{O}(|P||\Gamma|(|P|^2|\Delta| + |P||\Gamma|))$$

and space

$$\mathbf{O}(|P||\Gamma|(|P||\Delta| + |P||\Gamma|))$$

□

4.2.4 Shortest Lasso-Shaped Counterexamples

Repeating heads are not enough Assume again that $M = (P,\Gamma,\Delta,C_I)$ is a pushdown system such that $(p,\gamma,p',w') \in \Delta \Rightarrow |w'| \leq 2$. In his thesis [Sch02], Schwoon shows how to construct a shortest path to a reachable configuration. If applied to a pushdown system obtained by the transformation in Def. 3, the soonest second occurrence of a repeating head can be found. However, this is not sufficient to find shortest counterexamples.

Repeating prefixes Finding a shortest counterexample requires to extend the definition of a repeating head to a repeating prefix: any configuration $\langle p,w\rangle$ with $|w| > 0$ is a prefix. It is repeating iff there exist a path π and a word v with $\pi[0] = \langle p,w\rangle$ and $\pi[|\pi|-1] = \langle p,wv\rangle$.

Example For an example see the path in Fig. 4.3. The second occurrence of the repeating head (p_3,γ_0) can only be detected at $i = 23$ while the repeating prefix $\langle p_2,\gamma_2\gamma_1\gamma_0\gamma_2\gamma_1\gamma_0\rangle$ indicates a path whose heads form a lasso at $i = 18$.

Remarks The example also shows that the length of the prefix to be considered is $\mathbf{O}(|P||\Gamma|)$. On the other hand, once a path reaches a stack height of $|P||\Gamma|+1$ there must have been a second occurrence of a repeating head: consider an initialized path $\pi = \langle p_0, w_0\rangle \ldots \langle p_k, w_k\rangle$ such that $|w_k| = |P||\Gamma|+1$. Remember that the stack height grows or shrinks by at most one per transition. For each $0 \le h \le |P||\Gamma|+1$ there exists $0 \le i_h \le k$ such that all $\pi[i]$ with $i > i_h$ have stack height larger than h, i.e., $\forall i > i_h \,.\, |w_i| > |w_{i_h}| = h$. Clearly, there must be $h_1 \neq h_2$ such that $head(\pi[i_{h_1}]) = head(\pi[i_{h_2}])$. From the construction of the i_h, $\pi[0] \ldots \pi[i_{h_1}] \ldots \pi[i_{h_2}]$ provides evidence that $head(\pi[i_{h_1}])$ is a reachable repeatable head. As a final remark, it is clear that the length of any counterexample known to be present can be used to bound the length of a repeating prefix.

4.3 Timed Automata

4.3.1 Preliminaries

Notation is mostly from [CGP99]. For $x \in \mathbb{R}_0^+$, let $\lfloor x \rfloor$ and $fr(x)$ denote the integer and fractional parts of x.

A *timed word* over an alphabet Σ is a pair (α, τ) where $\alpha = \alpha[0]\alpha[1]\ldots$ is a (finite or infinite) word over Σ and τ is a non-decreasing sequence of time values $\tau[i] \in \mathbb{R}_0^+$ of the same length.

Let X be a set of clock variables ranging over $\mathbb{R}_0^+$. The set of *clock constraints* $\mathcal{C}(X)$ is defined as follows. If $x, y \in X$, $n \in \mathbb{N}_0$, and $\sim \in \{<, =, >\}$ then $x \sim n$ and $x - y \sim n$ are atomic clock constraints. If $\phi_1, \phi_2 \in \mathcal{C}(X)$ then also $\phi_1 \wedge \phi_2 \in \mathcal{C}(X)$.

A *clock assignment* for a set of clock variables X is a mapping $v : X \mapsto \mathbb{R}_0^+$. For $\lambda \subseteq X$, we define

$$v[\lambda := 0](x) = \begin{cases} 0 & \text{if } x \in \lambda \\ v(x) & \text{otherwise} \end{cases}$$

and for $d \in \mathbb{R}_0^+$

$$(v+d)(x) = v(x) + d$$

If no doubt can arise we also write x instead of $v(x)$. Satisfaction of a clock constraint ϕ by a clock assignment v, denoted $v \models \phi$, is defined in the natural way.

A *timed automaton* is a 6-tuple $A = (\Sigma, S, S_0, X, I, T)$ such that Σ is a finite *alphabet*, S is a finite set of *locations*, $S_0 \subseteq S$ is a set of *starting locations*, X is a finite set of *clocks*, $I : S \to \mathcal{C}(X)$ is a mapping from locations to clock constraints, called *location invariant*, and $T \subseteq S \times \Sigma \times \mathcal{C}(X) \times 2^X \times S$ is a set of *transition rules*.

A is *diagonal-free* iff all clock constraints are of the form $x \sim n$. For each $x \in X$, let its *ceiling* c_x denote the maximum n such that $x \sim n$ is a clock constraint in A. When appropriate, we also write c_i instead of c_{x_i} for some indexed clock x_i.

A *state* of A is a pair (s, v) such that $s \in S$ and v is a clock assignment to the clocks in X with $v \models I(s)$. (s, v) is *initial* iff $s \in S_0$ and $\forall x \in X \,.\, v(x) = 0$.

$(s, v) \overset{d,(a,\phi,\lambda)}{\to} (s', v')$ is a *transition* in A iff $(s, a, \phi, \lambda, s') \in T$, $d \in \mathbb{R}_0^+$, $v' = (v+d)[\lambda := 0]$, $\forall 0 \le d' \le d \,.\, v + d' \models I(s)$, $v + d \models \phi$, and $v' \models I(s')$.

A *path* of A over a timed word (α, τ) is a sequence $\pi : (s_0, v_0)(s_1, v_1)\dots$ with (s_i, v_i) states of A for all $0 \le i \le |\pi|$, such that

- $|\pi| = |\alpha| + 1$ if $|\alpha|$ is finite, $|\pi| = \infty$ otherwise,
- (s_0, v_0) is initial, and
- $\exists (s_0, \alpha[0], \phi_0, \lambda_0, s_1) \in T \;.\; (s_0, v_0) \overset{\tau[0],(\alpha[0],\phi_0,\lambda_0)}{\longrightarrow} (s_1, v_1)$ and
  $\forall 1 \le i < |\pi| - 1 \;.$
  $\exists (s_i, \alpha[i], \phi_i, \lambda_i, s_{i+1}) \in T \;.\; (s_i, v_i) \overset{\tau[i]-\tau[i-1],(\alpha[i],\phi_i,\lambda_i)}{\longrightarrow} (s_{i+1}, v_{i+1})$.

If $\pi = (s_0, v_0)(s_1, v_1)\dots$ is a path over (α, τ), we also write

$$(s_0, v_0) \overset{\alpha[0],\tau[0]}{\longrightarrow} (s_1, v_1) \overset{\alpha[1],\tau[1]}{\longrightarrow} \dots$$

Let v_1 and v_2 be two clock assignments. v_1 and v_2 are *region-equivalent*, $v_1 \cong v_2$, iff

- $\forall x \in X \;.\; v_1(x) > c_x \wedge v_2(x) > c_x \;\vee\; \lfloor v_1(x) \rfloor = \lfloor v_2(x) \rfloor$
- $\forall x \in X \;.\; (v_1(x) \le c_x) \Rightarrow (fr(v_1(x)) = 0 \Leftrightarrow fr(v_2(x)) = 0)$
- $\forall x, y \in X \;.\; (v_1(x) \le c_x \wedge v_1(y) \le c_y) \Rightarrow$
  $(fr(v_1(x)) \le fr(v_1(y)) \Leftrightarrow fr(v_2(x)) \le fr(v_2(y)))$

$[v_1]$ denotes the *equivalence class* of v_1 in $\cong$. If $\pi = (s_0, v_0)(s_1, v_1)\dots$ is a path in A we let $[\pi]$ denote $(s_0, [v_0])(s_1, [v_1])\dots$.

Alur and Dill showed [AD94] that region-equivalence can be used to construct a finite abstraction of a timed automaton that is sufficient for model checking of LTL formulae. Let A be a diagonal-free timed automaton. The *region automaton* of A, $R(A) = (S^R, S_0^R, T^R)$, is a finite automaton such that

$$\begin{array}{lcl} S^R & = & \{(s, [v]) \mid (s, v) \text{ is a state of } A\}, \\ S_0^R & = & \{(s_0, [v_0]) \mid (s_0, v_0) \text{ is initial}\} \subseteq S^R, \text{ and} \\ T^R & = & \{((s, [v]), a, (s', [v'])) \mid \\ & & \exists d \in \mathbb{R}_0^+, \exists (s, a, \phi, \lambda, s') \in T \;.\; (s, v) \overset{d,(a,\phi,\lambda)}{\longrightarrow} (s', v')\}. \end{array}$$

Lemma 15 *[CGP99] Let A be a diagonal-free timed automaton, let $R(A)$ be defined as above. A and $R(A)$ are bisimilar by $\{((s, v), (s, [v]))\}$.* □

Assume a transition with a guard that also contains Boolean *or*. Its clock constraint can be rewritten into disjunctive normal form and the transition can be split accordingly [BDGP98]. Having both, Boolean *and* and *or*, we can also add logical negation to the syntax of clock constraints: negations can be pushed inwards and the negation of an atomic constraint can be represented as disjunction of two atomic constraints. From now on we use arbitrary combinations of Boolean operators as well as relations from $\{<, \le, =, \ge, >\}$ in clock constraints on transitions. Following [BDGP98], we do not include strictness (i.e., the time component of a timed word must be strictly increasing) and non-Zenoness (the time component must be diverging) in our definition as these can be enforced by intersection with (fair) timed automata.

4.3.2 Reduction

Storing clock regions rather than clock valuations We adopt the finite abstraction to the region automaton in our reduction for timed automata and store the clock region rather than the exact valuation of the clocks.

Assume a set of clocks $X = \{x_0, \ldots, x_m\}$. A clock region over X can be represented by specifying [AD94]

1. for every clock $x_j \in X$ (the) one of the intervals $[0], (0;1), [1], \ldots, [c_j], (c_j;\infty)$ x_j is in, and
2. for every pair of clocks x_{j_1}, x_{j_2} such that $x_{j_1} \in (d; d+1)$ and $x_{j_2} \in (e; e+1)$ with $d+1 \leq c_{j_1}$ and $e+1 \leq c_{j_2}$, whether $fr(x_{j_1})$ is smaller, equal to, or greater than $fr(x_{j_2})$.

The representation of part (1) requires an integer variable p_j with range $0 \ldots c_j$ for each clock x_j. It holds the integral part of x_j, $\lfloor x_j \rfloor$, if $x_j \leq c_j$, or c_j otherwise. An additional bit $p_j^=$ per clock indicates whether $x_j = \lfloor x_j \rfloor$. For part (2) we use an array of integer variables $q_0 \ldots q_m$, each ranging from 0 to m, to store a permutation of the clock indices $0 \ldots m$.[§] The indices of all clocks x_j with $x_j \leq c_j$ are stored in the lower part of the array such that the fractional parts of the corresponding clocks increase. The upper part of the array stores the indices of all clocks x_j with $x_j > c_j$. Finally, m additional bits $q_j^=$ indicate whether the fractional parts of clocks $x_{q_{j-1}}, x_{q_j}$ are equal. Definition 4 formalizes the representation.

Definition 4 *Let v be a clock assignment for a set of clocks $X = \{x_0, \ldots, x_m\}$. $r2b(v)$ is a mapping of v to a set of representations of its region $[v]$ as follows:*

$$r2b : (X \mapsto \mathbb{R}_0^+) \mapsto 2^{\{0 \ldots c_0\} \times \ldots \times \{0 \ldots c_m\} \times \mathbb{B}^{m+1} \times \{0 \ldots m\}^{m+1} \times \mathbb{B}^m}$$

such that

$$\begin{aligned}
r2b(v) \quad = \quad & \{(p_0, \ldots, p_m, p_0^=, \ldots, p_m^=, q_0, \ldots, q_m, q_1^=, \ldots, q_m^=) \mid \\
& \left(\forall 0 \leq j \leq m \,.\, p_j = \begin{cases} \lfloor x_j \rfloor & \text{if } x_j < c_j \\ c_j & \text{otherwise} \end{cases} \right) \wedge \\
& (\forall 0 \leq j \leq m \,.\, p_j^= \Leftrightarrow x_j = p_j) \wedge \\
& (\forall 0 \leq j_1 \leq m \,.\, \exists 0 \leq j_2 \leq m \,.\, j_1 = q_{j_2}) \wedge \\
& (\forall 1 \leq j \leq m \,. \\
& \quad x_{q_{j-1}} \leq c_{q_{j-1}} \wedge x_{q_j} \leq c_{q_j} \wedge x_{q_{j-1}} - p_{q_{j-1}} \leq x_{q_j} - p_{q_j} \vee x_{q_j} > c_{q_j}) \wedge \\
& (\forall 1 \leq j \leq m \,.\, (q_j^= \Leftrightarrow x_{q_{j-1}} - p_{q_{j-1}} = x_{q_j} - p_{q_j}) \vee c_{q_j} < x_{q_j})\}
\end{aligned}$$

Note, that $r2b$ could be made canonical by imposing an order on q_{j-1}, q_j if either $x_{q_{j-1}} - p_{q_{j-1}} = x_{q_j} - p_{q_j}$ or $x_{q_{j-1}} > c_{q_{j-1}} \wedge x_{q_j} > c_{q_j}$ rather than mapping to sets. The following Lemma proves that regions are uniquely represented by disjoint sets of tuples. Hence, if two clock assignments can be represented by the same tuple, their regions are equal and vice versa.

Lemma 16 *Let v_1, v_2 be clock assignments over X. Then*

$$[v_1] = [v_2] \;\Rightarrow\; r2b(v_1) = r2b(v_2) \tag{4.1}$$

$$r2b(v_1) \cap r2b(v_2) \neq \emptyset \Rightarrow [v_1] = [v_2] \tag{4.2}$$

[§]The original analysis [AD94] requires $\mathbf{O}(\log((m+1)!)(m+1))$ bits whereas we use $\mathbf{O}(\log((m+1)^{m+1})(m+1))$ bits.

Proof: Follows from the definitions of region-equivalence and $r2b$. □

Formal definition ϵ-transitions strictly increase the power of timed automata [BDGP98]. Therefore, we do not introduce separate transitions to store or compare a state but combine this with existing transitions as in previous sections. If $(s,v) \stackrel{d,(a,\phi,\lambda)}{\rightarrow} (s',v')$ is a transition in A this leaves the choice of storing $(s,[v+d])$ or $(s',[(v+d)[\lambda := 0]])$. As in the finite and regular case we opt for the second variant. Definition 5 shows the reduction.

Definition 5 *Let* $A = (\Sigma, S, S_0, X, I, T)$ *with* $X = \{x_0, \ldots, x_m\}$ *be a diagonal-free timed automaton. Let* $\hat{s}_I \in S$ *be arbitrary but fixed, let* $\hat{p}_{0I} = \ldots = \hat{p}_{mI} = 0$, $\hat{p}^{=}_{0}{}_I \leftrightarrow \ldots \leftrightarrow \hat{p}^{=}_{mI} \leftrightarrow 1$, $\forall^m_{j=0} \,.\, \hat{q}_{jI} = j$, *and* $\hat{q}^{=}_{1}{}_I \leftrightarrow \ldots \leftrightarrow \hat{q}^{=}_{mI} \leftrightarrow 1$. *Then* $A^{\mathbf{S}} = (\Sigma, S^{\mathbf{S}}, {S_0}^{\mathbf{S}}, X, I^{\mathbf{S}}, T^{\mathbf{S}})$ *is defined as:*

$$
\begin{array}{lcll}
S^{\mathbf{S}} & = & \{(s,\hat{s},\hat{p}_0,\ldots,\hat{p}_m,\hat{p}^{=}_0,\ldots,\hat{p}^{=}_m,\hat{q}_0,\ldots,\hat{q}_m,\hat{q}^{=}_1,\ldots,\hat{q}^{=}_m, lo) \mid & \\
 & & \quad \forall 0 \le j_1 \le m \,.\, \exists 0 \le j_2 \le m \,.\, j_1 = \hat{q}_{j_2}\} & \\
 & \subseteq & S \times S \times \{0 \ldots c_0\} \times \ldots \times \{0 \ldots c_m\} \times \mathbb{B}^{m+1} \times \{0\ldots m\}^{m+1} \times \mathbb{B}^m \times \{st, lb, lc\} & \\
{S_0}^{\mathbf{S}} & = & \{(s_0,\hat{s}_I,\hat{p}_{0I},\ldots,\hat{p}_{mI},\hat{p}^{=}_0{}_I,\ldots,\hat{p}^{=}_{mI},\hat{q}_{0I},\ldots,\hat{q}_{mI},\hat{q}^{=}_1{}_I,\ldots,\hat{q}^{=}_{mI}, st) \mid s_0 \in S_0\} \cup & \\
 & & \{(s_0,s_0,0,\ldots,0,1,\ldots,1,\hat{q}_0,\ldots,\hat{q}_m,1,\ldots,1,lb) \mid s_0 \in S_0\} \subseteq S^{\mathbf{S}} & \\
I^{\mathbf{S}}(s^{\mathbf{S}}) & = & I(s) \quad \textit{where} \quad s^{\mathbf{S}} = (s,\hat{s},\hat{p}_0,\ldots,\hat{p}_m,\hat{p}^{=}_0,\ldots,\hat{p}^{=}_m,\hat{q}_0,\ldots,\hat{q}_m,\hat{q}^{=}_1,\ldots,\hat{q}^{=}_m, lo) & \\
T^{\mathbf{S}} & = & \{((s,\hat{s},\hat{p}_0,\ldots,\hat{p}_m,\hat{p}^{=}_0,\ldots,\hat{p}^{=}_m,\hat{q}_0,\ldots,\hat{q}_m,\hat{q}^{=}_1,\ldots,\hat{q}^{=}_m, lo), & \\
 & & \quad a, \Phi, \lambda, & \\
 & & \quad (s',\hat{s}',\hat{p}'_0,\ldots,\hat{p}'_m,\hat{p}^{=}_0{}',\ldots,\hat{p}^{=}_m{}',\hat{q}'_0,\ldots,\hat{q}'_m,\hat{q}^{=}_1{}',\ldots,\hat{q}^{=}_m{}', lo')) \mid & \\
 & & \quad (s,a,\phi,\lambda,s') \in T \,\wedge & \\
 & & \quad ((lo = st \wedge lo' = st \wedge \Psi_{\mathit{initial}} \wedge \Psi_{\mathit{unchanged}} \wedge (\Phi \equiv \phi)) \,\vee & (1) \\
 & & \quad\; (lo = st \wedge lo' = lb \wedge \Psi_{\mathit{initial}} \wedge s' = \hat{s}' \wedge (\Phi \equiv \phi \wedge \Phi_{\mathit{savecmp}})) \,\vee & (2) \\
 & & \quad\; (lo = lb \wedge lo' = lb \wedge \Psi_{\mathit{unchanged}} \wedge (\Phi \equiv \phi)) \,\vee & (3) \\
 & & \quad\; (lo = lb \wedge lo' = lc \wedge \Psi_{\mathit{unchanged}} \wedge s' = \hat{s}' \wedge (\Phi \equiv \phi \wedge \Phi_{\mathit{savecmp}})) \,\vee & (4) \\
 & & \quad\; (lo = lc \wedge lo' = lc \wedge \Psi_{\mathit{unchanged}} \wedge (\Phi \equiv \phi)))\} & (5)
\end{array}
$$

where

$$
\begin{array}{lcl}
\Psi_{\mathit{initial}} & \equiv & \hat{s} = \hat{s}_I \wedge (\bigwedge_{j=0}^m \hat{p}_j = \hat{p}_{jI} \wedge (\hat{p}^{=}_j \leftrightarrow \hat{p}^{=}_j{}_I) \wedge \hat{q}_j = \hat{q}_{jI}) \wedge (\bigwedge_{j=1}^m \hat{q}^{=}_j \leftrightarrow \hat{q}^{=}_j{}_I), \\
\Psi_{\mathit{unchanged}} & \equiv & \hat{s} = \hat{s}' \wedge (\bigwedge_{j=0}^m \hat{p}_j = \hat{p}_j{}' \wedge (\hat{p}^{=}_j \leftrightarrow \hat{p}^{=}_j{}') \wedge \hat{q}_j = \hat{q}_j{}') \wedge (\bigwedge_{j=1}^m (\hat{q}^{=}_j \leftrightarrow \hat{q}^{=}_j{}'))
\end{array}
$$

and

$$
\begin{array}{rcl}
\Phi_{savecmp} & \equiv & (\bigwedge_{j=0}^{m} \Phi_j^1 \wedge \Phi_j^{1=}) \wedge (\bigwedge_{j=1}^{m} \Phi_j^2 \wedge \Phi_j^{2=}) \\
& & \textit{with} \\
& & (\bigwedge_{j=0}^{m} (x_j \in \lambda \Rightarrow \hat{p}_j' = 0 \wedge (\Phi_j^1 \equiv 1)) \wedge \\
& & \quad (x_j \notin \lambda \Rightarrow (\hat{p}_j{}' < c_j \Rightarrow (\Phi_j^1 \equiv \hat{p}_j{}' \leq x_j < \hat{p}_j{}' + 1)) \wedge \\
& & \qquad (\hat{p}_j{}' = c_j \Rightarrow (\Phi_j^1 \equiv c_j \leq x_j)))) \wedge \\
& & (\bigwedge_{j=0}^{m} (x_j \in \lambda \Rightarrow \hat{p}_j^{=\prime} \wedge (\Phi_j^{1=} \equiv 1)) \wedge \\
& & \quad (x_j \notin \lambda \Rightarrow (\Phi_j^{1=} \equiv \hat{p}_j^{=\prime} \leftrightarrow x_j = \hat{p}_j{}'))) \wedge \\
& & (\bigwedge_{j=1}^{m} (x_{\hat{q}_{j-1}'} \in \lambda \Rightarrow (\Phi_j^2 \equiv 1)) \wedge \\
& & \quad (x_{\hat{q}_{j-1}'} \notin \lambda \wedge x_{\hat{q}_j'} \in \lambda \Rightarrow (\Phi_j^2 \equiv x_{\hat{q}_{j-1}'} - \hat{p}_{\hat{q}_{j-1}'}{}' = 0)) \wedge \\
& & \quad (x_{\hat{q}_{j-1}'} \notin \lambda \wedge x_{\hat{q}_j'} \notin \lambda \Rightarrow (\Phi_j^2 \equiv (x_{\hat{q}_{j-1}'} \leq c_{\hat{q}_{j-1}'} \wedge x_{\hat{q}_j'} \leq c_{\hat{q}_j'} \wedge \\
& & \qquad x_{\hat{q}_{j-1}'} - \hat{p}_{\hat{q}_{j-1}'}{}' \leq x_{\hat{q}_j'} - \hat{p}_{\hat{q}_j'}{}' \vee \\
& & \qquad c_{\hat{q}_j'} < x_{\hat{q}_j'})))) \wedge \\
& & (\bigwedge_{j=1}^{m} (x_{\hat{q}_j'} \in \lambda \Rightarrow \hat{q}_j^{=\prime} \wedge (\Phi_j^{2=} \equiv 1)) \wedge \\
& & \quad (x_{\hat{q}_{j-1}'} \in \lambda \wedge x_{\hat{q}_j'} \notin \lambda \Rightarrow (\Phi_j^{2=} \equiv (\hat{q}_j^{=\prime} \leftrightarrow x_{\hat{q}_j'} - \hat{p}_{\hat{q}_j'}{}' = 0) \vee c_{\hat{q}_j'} < x_{\hat{q}_j'})) \wedge \\
& & \quad (x_{\hat{q}_{j-1}'} \notin \lambda \wedge x_{\hat{q}_j'} \notin \lambda \Rightarrow (\Phi_j^{2=} \equiv (\hat{q}_j^{=\prime} \leftrightarrow x_{\hat{q}_{j-1}'} - \hat{p}_{\hat{q}_{j-1}'}{}' = x_{\hat{q}_j'} - \hat{p}_{\hat{q}_j'}{}') \vee \\
& & \qquad c_{\hat{q}_j'} < x_{\hat{q}_j'})))
\end{array}
$$

Correctness Lemma 17 states that A^{S} can make a transition that stores or compares a state of A iff the added location bits of A^{S} represent the location and the region of the clock assignment of the A-state in the post-state.

Lemma 17 *Let $A = (\Sigma, S, S_0, X, I, T)$ be a timed automaton and A^{S} be defined as above. Then*

(i)
$$
\begin{array}{c}
((s, \hat{s}_I, \hat{p}_{0I}, \ldots, \hat{p}_{mI}, \hat{p}_0^{=}{}_I, \ldots, \hat{p}_{mI}^{=}, \hat{q}_{0I}, \ldots, \hat{q}_{mI}, \hat{q}_1^{=}{}_I, \ldots, \hat{q}_{mI}^{=}, st), v) \overset{d,(a,\Phi,\lambda)}{\longrightarrow} \\
((s', s', \hat{p}_0', \ldots, \hat{p}_m', \hat{p}_0^{=\prime}, \ldots, \hat{p}_m^{=\prime}, \hat{q}_0', \ldots, \hat{q}_m', \hat{q}_1^{=\prime}, \ldots, \hat{q}_m^{=\prime}, lb), v') \\
\Leftrightarrow \\
(s, v) \overset{d,(a,\phi,\lambda)}{\longrightarrow} (s', v') \wedge \\
(\hat{p}_0', \ldots, \hat{p}_m', \hat{p}_0^{=\prime}, \ldots, \hat{p}_m^{=\prime}, \hat{q}_0', \ldots, \hat{q}_m', \hat{q}_1^{=\prime}, \ldots, \hat{q}_m^{=\prime}) \in r2b(v')
\end{array}
$$

(ii)
$$
\begin{array}{c}
((s, s', \hat{p}_0{}', \ldots, \hat{p}_m{}', \hat{p}_0^{=\prime}, \ldots, \hat{p}_m^{=\prime}, \hat{q}_0{}', \ldots, \hat{q}_m{}', \hat{q}_1^{=\prime}, \ldots, \hat{q}_m^{=\prime}, lb), v) \overset{d,(a,\Phi,\lambda)}{\longrightarrow} \\
((s', s', \hat{p}_0{}', \ldots, \hat{p}_m{}', \hat{p}_0^{=\prime}, \ldots, \hat{p}_m^{=\prime}, \hat{q}_0{}', \ldots, \hat{q}_m{}', \hat{q}_1^{=\prime}, \ldots, \hat{q}_m^{=\prime}, lc), v') \\
\Leftrightarrow \\
(s, v) \overset{d,(a,\phi,\lambda)}{\longrightarrow} (s', v') \wedge \\
(\hat{p}_0{}', \ldots, \hat{p}_m{}', \hat{p}_0^{=\prime}, \ldots, \hat{p}_m^{=\prime}, \hat{q}_0{}', \ldots, \hat{q}_m{}', \hat{q}_1^{=\prime}, \ldots, \hat{q}_m^{=\prime}) \in r2b(v')
\end{array}
$$

Proof: Follows from Def. 4 and 5 with

$$
(s, v) \overset{0,(a,\phi,\lambda)}{\longrightarrow} (s', v') \quad \Rightarrow \quad \forall x \in X \,.\, x \in \lambda \wedge x' = 0 \vee x \notin \lambda \wedge x' = x
$$

□

Now we can prove the existence of an infinite loop in A iff A^{S} can reach a state such that $lo = lc$.

Theorem 18 *Let $A = (\Sigma, S, S_0, X, I, T)$ be a timed automaton, A^{S} be defined as above, and $k > l \geq 0$. A has a path $\pi = (s_0, v_0)(s_1, v_1)\ldots$ such that*

$$[\pi] \;=\; (s_0, [v_0]) \stackrel{\alpha[0]}{\rightarrow} \ldots \stackrel{\alpha[l-2]}{\rightarrow} (s_{l-1}, [v_{l-1}]) \stackrel{\alpha[l-1]}{\rightarrow} ((s_l, [v_l]) \stackrel{\alpha[l]}{\rightarrow} \ldots \stackrel{\alpha[k-2]}{\rightarrow} (s_{k-1}, [v_{k-1}]) \stackrel{\alpha[k-1]}{\rightarrow})^{\omega}$$

if and only if A^{S} has a path

$$\begin{array}{rcll} \pi^{\mathrm{S}} & = & ((s_0, \hat{s}_I, \hat{p}_{0I}, \ldots, \hat{p}_{mI}, \hat{p}_0^{=}{}_I, \ldots, \hat{p}_{mI}^{=}, \hat{q}_{0I}, \ldots, \hat{q}_{mI}, \hat{q}_1^{=}{}_I, \ldots, \hat{q}_{mI}^{=}, st), v_0) & \stackrel{\alpha[0]}{\rightarrow} \\ & & \ldots & \stackrel{\alpha[l-2]}{\rightarrow} \\ & & ((s_{l-1}, \hat{s}_I, \hat{p}_{0I}, \ldots, \hat{p}_{mI}, \hat{p}_0^{=}{}_I, \ldots, \hat{p}_{mI}^{=}, \hat{q}_{0I}, \ldots, \hat{q}_{mI}, \hat{q}_1^{=}{}_I, \ldots, \hat{q}_{mI}^{=}, st), v_{l-1}) & \stackrel{\alpha[l-1]}{\rightarrow} \\ & & ((s_l, s_l, \hat{p}_0, \ldots, \hat{p}_m, \hat{p}_0^{=}, \ldots, \hat{p}_m^{=}, \hat{q}_0, \ldots, \hat{q}_m, \hat{q}_1^{=}, \ldots, \hat{q}_m^{=}, lb), v_l) & \stackrel{\alpha[l]}{\rightarrow} \\ & & \ldots & \stackrel{\alpha[k-2]}{\rightarrow} \\ & & ((s_{k-1}, s_l, \hat{p}_0, \ldots, \hat{p}_m, \hat{p}_0^{=}, \ldots, \hat{p}_m^{=}, \hat{q}_0, \ldots, \hat{q}_m, \hat{q}_1^{=}, \ldots, \hat{q}_m^{=}, lb), v_{k-1}) & \stackrel{\alpha[k-1]}{\rightarrow} \\ & & ((s_k, s_l, \hat{p}_0, \ldots, \hat{p}_m, \hat{p}_0^{=}, \ldots, \hat{p}_m^{=}, \hat{q}_0, \ldots, \hat{q}_m, \hat{q}_1^{=}, \ldots, \hat{q}_m^{=}, lc), v_k) & \end{array}$$

Proof: "$\Rightarrow$": Assume $l > 0$. Set $\pi^{\mathrm{S}}[0] = (s_0, \ldots, st)$ according to Def. 5. $\pi^{\mathrm{S}}[0]\ldots\pi^{\mathrm{S}}[l-1]$ can be constructed from π by taking transitions from (1). Using Lemma 17 we have the transition of A^{S} from $\pi^{\mathrm{S}}[l-1]$ to $\pi^{\mathrm{S}}[l]$ where $(\hat{p}_0, \ldots, \hat{p}_m, \hat{p}_0^{=}, \ldots, \hat{p}_m^{=}, \hat{q}_0, \ldots, \hat{q}_m, \hat{q}_1^{=}, \ldots, \hat{q}_m^{=}) \in r2b(v_l)$. Transitions in (3) lead to $\pi^{\mathrm{S}}[k-1]$. By assumption $s_l = s_k$ and $[v_l] = [v_k]$. With Lemma 17 this gives the transition from $\pi^{\mathrm{S}}[k-1]$ to $\pi^{\mathrm{S}}[k]$. If $l = 0$, choose an initial state $\pi^{\mathrm{S}}[l]$ such that the stored location is s_l, the stored region is the initial region, and $lo = lb$. Continue with transitions from sets (3) and (4) as for $l > 0$.

"$\Leftarrow$": Assume π^{S} with $k > l$. By construction of A^{S} there exists a path

$$\pi' \;=\; (s_0, v_0) \stackrel{\alpha[0]}{\rightarrow} \ldots \stackrel{\alpha[l-2]}{\rightarrow} (s_{l-1}, v_{l-1}) \stackrel{\alpha[l-1]}{\rightarrow} (s_l, v_l) \stackrel{\alpha[l]}{\rightarrow} \ldots \stackrel{\alpha[k-2]}{\rightarrow} (s_{k-1}, v_{k-1}) \stackrel{\alpha[k-1]}{\rightarrow} (s_k, v_k)$$

in A. With Lemma 15 there is a path

$$\begin{array}{rcl} \pi'_R & = & (s_0, [v_0]) \stackrel{\alpha[0]}{\rightarrow} \ldots \stackrel{\alpha[l-2]}{\rightarrow} (s_{l-1}, [v_{l-1}]) \stackrel{\alpha[l-1]}{\rightarrow} \\ & & (s_l, [v_l]) \stackrel{\alpha[l]}{\rightarrow} \ldots \stackrel{\alpha[k-2]}{\rightarrow} (s_{k-1}, [v_{k-1}]) \stackrel{\alpha[k-1]}{\rightarrow} (s_k, [v_k]) \end{array}$$

in $R(A)$. From the construction of A^{S}, we have $s_l = s_k$, and, with Lemmas 16 and 17, $[v_l] = [v_k]$. Hence,

$$\pi''_R \;=\; (s_0, [v_0]) \stackrel{\alpha(0)}{\rightarrow} \ldots \stackrel{\alpha[l-2]}{\rightarrow} (s_{l-1}, [v_{l-1}]) \stackrel{\alpha[l-1]}{\rightarrow} ((s_l, [v_l]) \stackrel{\alpha[l]}{\rightarrow} \ldots \stackrel{\alpha[k-2]}{\rightarrow} (s_{k-1}, [v_{k-1}]) \stackrel{\alpha[k-1]}{\rightarrow})^{\omega}$$

is an infinite path in $R(A)$, which by Lemma 15 gives π in A as required. $\square$

Remark Note that, while A is assumed to be diagonal-free, A^{S} might not. This is purely a matter of convenience. For a reduction from timed automata to diagonal-free timed automata see [BDGP98]. We could also admit difference constraints in the construction of A^{S} similar to that reduction.

4.3.3 Complexity

Theorem 19 *Let $A = (\Sigma, S, S_0, X, I, T)$ be a timed automaton. $A^{\mathbf{S}}$ has*

$$\mathbf{O}((\Pi_{j=0}^{|X|-1} (c_j + 1)) \cdot |X|! \cdot 2^{2|X|} \cdot |S| \cdot |S|) \text{ locations}$$

and

$$\mathbf{O}((\Pi_{j=0}^{|X|-1} (c_j + 1)) \cdot |X|! \cdot 2^{2|X|} \cdot |S| \cdot |T|) \text{ transitions.}$$

The number of clock regions is equal to those of A.

Proof: The locations of A are extended to store another location and a clock region. There are $\mathbf{O}((\Pi_{j=0}^{|X|-1} (c_j + 1)) \cdot |X|! \cdot 2^{2|X|})$ of the latter [AD94]. The transformation adds at most $|X|$ constraints of the form $x_{j_1} - x_{j_2} \sim c$. Making $A^{\mathbf{S}}$ diagonal-free therefore adds another $\mathbf{O}(2^{|X|})$ location bits [BDGP98]. The number of transitions can be derived in a similar way as in Thm. 6.
□

4.3.4 Shortest Lasso-Shaped Counterexamples

As in the finite case (Sect. 3.4) the reduction can be used to find lasso-shaped counterexamples with a minimal number of transitions for LTL properties if breadth-first search is used to determine reachability. Alternatively, using a priority queue instead of a queue in the reachability algorithm, the lasso-shaped path that spends least time until the closure of the loop can be found [BFH⁺01]. UPPAAL [LPY97] offers both possibilities.

4.4 Related Work

Reduction to reachability Bouajjani et al. independently used the same reduction to verify liveness properties in regular model checking [BHV04]. They only sketch the reduction. No complexity results are given and timed automata are not discussed. The reduction of Shilov et. al [SYE⁺05] applies in principle also to infinite states systems if their prerequisites are satisfied. However, they do not give concrete examples.

Restriction to reachability Aceto et al. [ABBL03] developed a specification language for timed systems and proved for a subset that it can express precisely those properties that can be checked by reachability in the timed system composed with a test automaton (basically, a timed automaton with designated bad locations).

Proving termination with transition invariants Podelski and Rybalchenko use invariants of the transition relation rather than of the set of reachable states to establish liveness properties of infinite state programs: roughly, a program satisfies a liveness property iff there exists a disjunctively well-founded invariant of its (suitably restricted) transition relation [PR04]. In [PR05, CPR05] they continue by applying predicate abstraction and counterexample-guided abstraction refinement to transitions rather than states. Instead of working with transitions directly one could apply the part of the state-recording translation that non-deterministically saves a state and then work with state-based invariants or abstractions. As liveness properties of infinite state systems do not necessarily have lasso-shaped counterexamples, our simple loop-closing condition would have to be replaced with a well-foundedness check as used by [PR04, PR05, CPR05].

Other Early work on liveness for regular model checking includes [BJNT00, PS00]. Pnueli and Shahar [PS00] also use a copy of a current state to detect bad cycles in parameterized systems. However, this is not performed as syntactic transformation of a system but as part of a dedicated liveness checking algorithm. A variant of LTL geared towards parameterized systems is proposed in [AJN+04]. [BLW04b] gives details on how to encode a broader set of properties than [AJN+04] for ($\omega-$)regular model checking, which can be used in conjunction with our reduction. Algorithms to compute repeated reachability, on which we also base our reductions, can be found for pushdown systems, e.g., in [BEM97], and for timed automata in [AD94].

4.5 Summary

We have extended the state-recording translation to some infinite state systems. Saving and comparing configurations in regular model checking can be achieved by working with automata on tuples of characters rather than individual characters. The transformation preserves bounded local depth, which is a sufficient criterion for termination of an algorithm to compute the transitive closure of the transition relation. Pushdown systems can be handled by storing the current head and marking the stack height. When the saved head occurs a second time without the stack height falling below the level at the time of saving, a loop has been detected. For timed automata the region abstraction of a configuration is saved rather than the (infinite) configuration itself. Note, that in all cases an existing algorithm to verify liveness for the respective class of systems has been syntactically expressed in that class of systems. While for timed automata the transformation can help to find lasso-shaped counterexamples with the least number of transitions, finding shortest counterexamples for pushdown systems would require detection of repeating prefixes.

5

Büchi Automata for Shortest Counterexamples

The present letter is a very long one, simply because I had no leisure to make it shorter.
Blaise Pascal, Provincial Letters: Letter XVI

In the automaton-based approach to model checking, a PLTLB property is verified by searching for loops in the synchronous product of a Kripke structure M, representing the model, and a Büchi automaton B, accepting counterexamples for the property. To obtain shortest counterexamples, B must be able to accept these in a "short enough" way. In this chapter we formally define, when a Büchi automaton accepts shortest counterexamples (termed *tight*), and we present necessary and sufficient conditions for tightness (Sect. 5.1). Section 5.2 examines whether existing approaches meet these conditions. It turns out that none of the constructions we looked at fulfills the criteria for PLTLB. Therefore, in Sect. 5.3 we give a construction of a tight Büchi automaton from a PLTLB formula. The construction is generalized to tighten arbitrary Büchi automata in Sect. 5.4. Section 5.5 discusses related work and Sect. 5.6 gives a brief summary.

5.1 Tight Büchi Automata

Intuition If shortest counterexamples are desired in the automaton-based approach to model checking, the product of the model M and the Büchi automaton B must have an initialized fair path $\lambda = \langle \mu, \nu \rangle$ that can be represented as lasso of the same length as the shortest counterexample $\alpha = \langle \beta, \gamma \rangle$ in M. From Lemma 1 it can be inferred that a path $\lambda = \langle \mu, \nu \rangle$ in $M \times B$ of the same length as the counterexample $\alpha = \langle \beta, \gamma \rangle$ in M implies that the corresponding path $\rho = \sigma\tau^\omega$ in B can be represented as the same type as $\langle \beta, \gamma \rangle$. In other words, the Büchi automaton should adapt as a chameleon to the counterexamples present in the model.

Example Consider the scenarios in Fig. 5.1. The Büchi automaton B in the left scenario has a path $\sigma\tau^\omega$ of the same structure as the counterexample $\beta\gamma^\omega$ in M, leading to an equally short counterexample $(\beta \times \sigma)(\gamma \times \tau)^\omega$ in the product $M \times B$. The path of the Büchi automaton in the right scenario has an unnecessarily long stem and loop. Note, that the length of the stem in $M \times B$ is the maximum of the lengths of the stems in M and B, and the length of the loop in $M \times B$ is the least common multiple of the lengths of the loops in M and B.

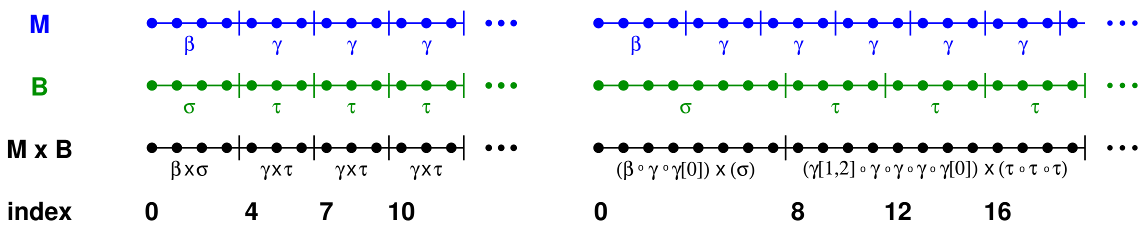


Figure 5.1: Scenarios with shortest and non-optimal counterexample

Formal definition Kupferman and Vardi [KV01] call an automaton on finite words *tight* if it accepts shortest prefixes for violations of safety formulae. We extend that notion to Büchi automata on infinite words.

Definition 6 *Let $B = (S, T, I, L, F)$ be a Büchi automaton. B is* tight *iff*

$$\forall \alpha \in Lang(B) \;.\; \forall \beta, \gamma \;.\; (\alpha = \beta\gamma^\omega \Rightarrow \\ \exists \rho \in \Pi^F(B) \;.\; (L(\rho) = \alpha \wedge type(\langle \beta, \gamma \rangle) \in type(\rho)))$$

Alternative criteria The left scenario in Fig. 5.1 suggests another, alternative formulation, which may be more intuitive and is easier to prove for some automata: the subsequences of α starting at indices $4, 7, 10, \ldots$ are the same, as are those beginning at $5, 8, 11, \ldots$, and $6, 9, 12, \ldots$. On the other hand, the subsequences starting at the respective indices in a single iteration (e.g., 4, 5, 6) are all different — otherwise a part of the loop could be cut out, contradicting minimality. Hence, if B is tight, there must be an initialized fair path ρ with $L(\rho) = \alpha$ with the following property: for each pair of indices i, j, if the subsequences of α starting at i and j have the same future ($\alpha[i, \infty] = \alpha[j, \infty]$), then ρ maps i and j to the same state in B ($\rho[i] = \rho[j]$). Theorem 20 establishes the equivalence of the criteria.

Theorem 20 *Let $B = (S, T, I, L, F)$ be a Büchi automaton. The following statements are equivalent:*

1. *B is tight.*

2. $\forall \alpha \in Lang(B) \;.\; \forall \beta, \gamma \;.\; (\langle \beta, \gamma \rangle$ *is minimal for* $\alpha \Rightarrow$
   $\exists \rho \in \Pi^F(B) \;.\; (L(\rho) = \alpha \wedge type(\langle \beta, \gamma \rangle) \in type(\rho)))$

3. $\forall \alpha \in Lang(B) \;.\; ((\exists \beta, \gamma \;.\; \alpha = \beta\gamma^\omega) \Rightarrow$
   $(\exists \rho \in \Pi^F(B) \;.\; (L(\rho) = \alpha \wedge (\forall i, j \;.\; \alpha[i, \infty] = \alpha[j, \infty] \Rightarrow \rho[i] = \rho[j]))))$

4. $\forall \alpha \in Lang(B) \;.\; \forall \beta, \gamma \;.\; (\langle \beta, \gamma \rangle$ *is minimal for* $\alpha \Rightarrow$
   $\exists \rho \in \Pi^F(B) \;.\; \exists \sigma, \tau \;.\; (L(\rho) = \alpha \;\; \wedge \;\; \rho = \sigma\tau^\omega \;\; \wedge \;\; |\langle \sigma, \tau \rangle| = |\langle \beta, \gamma \rangle|))$

5. $\forall \alpha \in Lang(B) \;.\; \forall \beta, \gamma \;.\; (\alpha = \beta\gamma^\omega \Rightarrow$
   $\exists \rho \in \Pi^F(B) \;.\; \exists \sigma, \tau \;.\; (L(\rho) = \alpha \;\; \wedge \;\; \rho = \sigma\tau^\omega \;\; \wedge \;\; |\langle \sigma, \tau \rangle| = |\langle \beta, \gamma \rangle|))$

Proof: 5 $\Rightarrow$ 4: Obvious, $\langle \beta, \gamma \rangle$ being minimal for α implies $\alpha = \beta\gamma^\omega$.

$4 \Rightarrow 3$: Assume $\alpha \in \mathit{Lang}(B)$ with $\langle\beta,\gamma\rangle$ minimal for α. Hence, there exists a path ρ in $\Pi^F(B)$ with $L(\rho) = \alpha$ and σ, τ such that $\rho = \sigma\tau^\omega$ with $|\langle\sigma,\tau\rangle| = |\langle\beta,\gamma\rangle|$. Corollary 3 gives $|\sigma| = |\beta|$ and $|\tau| = |\gamma|$. Let i, j with $\alpha[i,\infty] = \alpha[j,\infty]$. It remains to show that $\rho[i] = \rho[j]$. This is done by distinguishing 5 cases according to the positions of i and j w.r.t. to β and γ in α. Note that only in the first and in the last case $\rho[i]$ and $\rho[j]$ actually play a role as in all other cases $\langle\beta,\gamma\rangle$ cannot be minimal for α.

Case 1, $i = j$: Obvious.

Case 2, $i < j \leq |\beta| - 1$:

$$\begin{aligned} \alpha[i,\infty] = \alpha[j,\infty] &\Rightarrow \alpha[0,i-1] \circ \alpha[i,\infty] = \alpha[0,i-1] \circ \alpha[j,\infty] \\ &\Rightarrow \alpha = \beta[0,i-1] \circ \beta[j,|\beta|-1] \circ \gamma^\omega \\ &\Rightarrow \text{contradiction, } \langle\beta,\gamma\rangle \text{ is minimal for } \alpha \end{aligned}$$

Case 3, $|\beta| \leq i < j < |\beta| + |\gamma|$:

$$\begin{aligned} \alpha[i,\infty] = \alpha[j,\infty] &\Rightarrow \alpha[0,i-1] \circ \alpha[i,\infty] = \alpha[0,i-1] \circ \alpha[j,\infty] \\ &\Rightarrow \alpha = \beta \circ (\gamma[0,i-1-|\beta|] \circ \gamma[j-|\beta|,|\gamma|-1])^\omega \\ &\Rightarrow \text{contradiction, } \langle\beta,\gamma\rangle \text{ is minimal for } \alpha \end{aligned}$$

Case 4, $0 \leq i < |\beta| \leq j < |\beta| + |\gamma|$:

$$\begin{aligned} \alpha[i,\infty] = \alpha[j,\infty] &\Rightarrow \alpha[0,i-1] \circ \alpha[i,\infty] = \alpha[0,i-1] \circ \alpha[j,\infty] \\ &\Rightarrow \alpha = \beta[0,i-1] \circ (\gamma[j-|\beta|,|\gamma|-1] \circ \gamma[0,j-|\beta|-1])^\omega \\ &\Rightarrow \text{contradiction, } \langle\beta,\gamma\rangle \text{ is minimal for } \alpha \end{aligned}$$

Case 5, $|\beta| + |\gamma| \leq i$ and/or j: Clearly, if $i \geq |\beta| + |\gamma|$, then $\alpha[i,\infty] = \alpha[i - |\gamma|, \infty]$. Hence, reduce to 1 – 4 by subtracting $|\gamma|$ from i and/or j. In cases 2 – 4 we can stop. For case 1, remember that $|\sigma| = |\beta|$ and $|\tau| = |\gamma|$; therefore, $\rho[i] = \rho[i - |\gamma|]$ for any $i \geq |\beta| + |\gamma|$.

$3 \Rightarrow 2$: Assume $\alpha = \beta\gamma^\omega \in \mathit{Lang}(B)$ and ρ a path in $\Pi^F(B)$ with $L(\rho) = \alpha$ and $\forall i, j \,.\, \alpha[i,\infty] = \alpha[j,\infty] \Rightarrow \rho[i] = \rho[j]$. Let $\langle\beta,\gamma\rangle$ be minimal for α.

$$\begin{aligned} \alpha = \beta\gamma^\omega &\Rightarrow \forall i < |\gamma|, \forall k \,.\, \alpha[|\beta| + i, \infty] = (\gamma^\omega)[i,\infty] = \alpha[|\beta| + i + |\gamma|k, \infty] \\ &\Rightarrow \forall i < |\gamma|, \forall k \,.\, \rho[|\beta| + i] = \rho[|\beta| + i + |\gamma|k] \end{aligned}$$

Let $\sigma = \rho[0, |\beta| - 1]$ and $\tau = \rho[|\beta|, |\beta| + |\gamma| - 1]$. Hence, $\rho = \sigma\tau^\omega$ such that $|\sigma| = |\beta|$ and $|\tau| = |\gamma|$.

$2 \Rightarrow 1$: Assume $\alpha = \beta\gamma^\omega \in \mathit{Lang}(B)$. Let $\langle\beta',\gamma'\rangle$ be minimal for α and ρ a path in $\Pi^F(B)$ with $L(\rho) = \alpha$ and $\mathit{type}(\langle\beta',\gamma'\rangle) \in \mathit{type}(\rho)$. Hence, there exist σ', τ' with $\rho = \sigma'\tau'^\omega$, $|\sigma'| = |\beta'|$, and $|\tau'| = |\gamma'|$. Lemma 2 gives $|\beta| \geq |\beta'|$ and $|\gamma'|$ divides $|\gamma|$. With $\sigma = \rho[0, |\beta| - 1]$ and $\tau = \rho[|\beta|, |\beta| + |\gamma| - 1]$ we have $\rho = \sigma\tau^\omega$ and $\mathit{type}(\langle\beta,\gamma\rangle) \in \mathit{type}(\rho)$.

$1 \Rightarrow 5$: Assume $\alpha = \beta\gamma^\omega \in \mathit{Lang}(B)$. Let ρ be a path in $\Pi^F(B)$ with $L(\rho) = \alpha$ and $\mathit{type}(\langle\beta,\gamma\rangle) \in \mathit{type}(\rho)$. Hence, there exist σ, τ with $\rho = \sigma\tau^\omega$, $|\sigma| = |\beta|$, and $|\tau| = |\gamma|$. By definition of length of a lasso, $|\langle\sigma,\tau\rangle| = |\langle\beta,\gamma\rangle|$. □

Basic facts The following propositions show general ways to obtain a tight Büchi automaton. Both, the sum and the product, which correspond to language union and intersection, of two tight Büchi automata are also tight. The third proposition suggests a saturation procedure. A tight automaton is obtained by

1. adding states to a Büchi automaton such that for each pair of states s_i, s_j there is a third state accepting the intersection of the languages of s_i and s_j,

2. adding a transition from s_i to s_j for each pair of states s_i, s_j if the language of s_j is a subset of the language of s_i with the first character chopped of, and, finally,

3. making every state initial that accepts a subset of the language of the automaton.

Proposition 21 *Let $B_1 = (S_1, T_1, I_1, L_1, F_1)$, $B_2 = (S_2, T_2, I_2, L_2, F_2)$ be two tight Büchi automata. Then $B_3 = B_1 + B_2$ is tight.*

Proof: Let $\alpha = \langle \beta, \gamma \rangle \in Lang(B_3)$ such that $\langle \beta, \gamma \rangle$ is minimal for α. By construction of B_3, either 1) $\alpha|_{AP_1} \in Lang(B_1)$ or 2) $\alpha|_{AP_2} \in Lang(B_2)$. Assume 1). There exists a path $\rho_1 = \sigma_1 \tau_1^\omega \in \Pi^F(B_1)$ with $|\sigma_1| = |\beta|$, $|\tau_1| = |\gamma|$, and $L(\rho_1) = \alpha|_{AP_1}$. By definition of the sum of automata $\rho_3 = ((\sigma_1 \times \sigma_2), (\tau_1 \times \tau_2)) \in \Pi^F(B_3)$ with $\sigma_2 = \beta|_{AP_2 \setminus AP_1}$ and $\tau_2 = \gamma|_{AP_2 \setminus AP_1}$ is a path in $\Pi(B_3)$ s.t. $L(\rho_3) = \alpha$. Clearly, $|\sigma_1 \times \sigma_2| = |\beta|$ and $|\tau_1 \times \tau_2| = |\gamma|$. Case 2) is analogous. □

Proposition 22 *Let $B_1 = (S_1, T_1, I_1, L_1, F_1)$, $B_2 = (S_2, T_2, I_2, L_2, F_2)$ be two tight Büchi automata. Then $B_3 = B_1 \times B_2$ is tight.*

Proof: Let $\alpha = \langle \beta, \gamma \rangle \in Lang(B_3)$ such that $\langle \beta, \gamma \rangle$ is minimal for α. By Lemma 3 $\langle \beta, \gamma \rangle$ is unique. Since $\alpha|_{AP_1} \in Lang(B_1)$ and $\alpha|_{AP_2} \in Lang(B_2)$, there exist paths $\rho_1 = \sigma_1 \tau_1^\omega \in \Pi^F(B_1)$ and $\rho_2 = \sigma_2 \tau_2^\omega \in \Pi^F(B_2)$ with $|\sigma_1| = |\sigma_2| = |\beta|$ and $|\tau_1| = |\tau_2| = |\gamma|$. By definition of the synchronous product there is a path $\rho_3 = ((\sigma_1 \times \sigma_2), (\tau_1 \times \tau_2)) \in \Pi^F(B_3)$ with $L(\rho_3) = \alpha$. Clearly, $|\sigma_1 \times \sigma_2| = |\beta|$ and $|\tau_1 \times \tau_2| = |\gamma|$. □

Proposition 23 *Let $B = (S, T, I, L, F)$ be a Büchi automaton. B is tight if*

$$(\forall s_1, s_2 \in S \,.\, \exists s_3 \in S \,.\, Lang(B, s_1) \cap Lang(B, s_2) = Lang(B, s_3)) \quad \wedge \qquad (1)$$
$$(\forall s_1, s_2 \in S \,.\, Lang(B, s_2) \subseteq \{\alpha[1, \infty] \mid \alpha \in Lang(B, s_1)\} \Rightarrow (s_1, s_2) \in T) \quad \wedge \qquad (2)$$
$$(\forall s \in S \,.\, Lang(B, s) \subseteq Lang(B) \Rightarrow s \in I) \qquad (3)$$

Proof: Let $\alpha \in Lang(B)$. By (1), for each position i in α there exists at least one minimal state $s_i^{\min}$ with $\alpha[i, \infty] \in Lang(B, s_i^{\min})$ and $\forall s' \in S \,.\, \alpha[i, \infty] \in Lang(B, s') \Rightarrow Lang(B, s_i^{\min}) \subseteq Lang(B, s_i)$. As $\alpha[i, \infty] \in Lang(B, s_i^{\min})$, there is s'_{i+1} with $(s_i^{\min}, s'_{i+1}) \in T$ and $\alpha[i+1, \infty] \in Lang(B, s'_{i+1})$. Clearly, $Lang(B, s_{i+1}^{\min}) \subseteq Lang(B, s'_{i+1}) \subseteq \{\alpha'[1, \infty] \mid \alpha' \in Lang(B, s_i^{\min})\}$. Hence, with (2), $(s_i^{\min}, s_{i+1}^{\min}) \in T$. Further, with (1) and (3), $s_0^{\min}$ is an initial state. By choosing the same state $s^{\min}$ for any i, j such that $\alpha[i, \infty] = \alpha[j, \infty]$ we can construct a path ρ in $\Pi^F(B)$ with $L(\rho) = \alpha$ and $\alpha[i, \infty] = \alpha[j, \infty] \Rightarrow \rho[i] = \rho[j]$. Tightness follows from Thm. 20. □

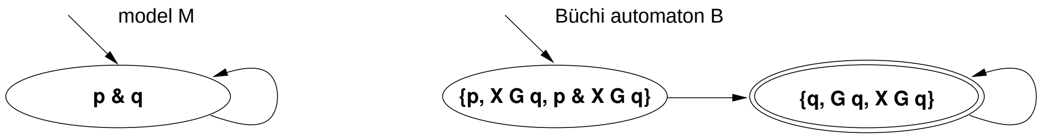


Figure 5.2: Model M and Büchi automaton $B^{p\wedge\mathbf{XG}q}_{GPVW}$ resulting in non-optimal counterexample

5.2 (Non-) Optimality of Specific Approaches

5.2.1 Gerth et al. (GPVW)

Motivation and example The approach by Gerth et al. (GPVW) [GPVW96] for future time LTL forms the basis of many algorithms to construct small Büchi automata, which benefits explicit state model checking but is also used, e.g., for symbolic model checking in VIS [VIS96]. Figure 5.2 shows an example that GPVW does not, in general, lead to tight automata. The model, M, has a single state with propositions p, q true. $B^{p\wedge\mathbf{XG}q}_{GPVW}$ accepts counterexamples to $\neg(p \wedge \mathbf{XG}q)$. Its states are labeled with the content of the Old-set*; the $Next$-set is $\{\mathbf{G}q\}$ for both nodes. Paths starting from the initial state of $B^{p\wedge\mathbf{XG}q}_{GPVW}$ fulfill $p \wedge \mathbf{XG}q$, those starting from the fair state satisfy $\mathbf{G}q$. M has a single, infinite path satisfying $\mathbf{G}(p \wedge q)$. While this is a counterexample of length 1, the shortest initialized fair lasso in the product $M \times B^{p\wedge\mathbf{XG}q}_{GPVW}$ has length 2. Note that adding transitions or designating more initial states is not enough to make $B^{p\wedge\mathbf{XG}q}_{GPVW}$ tight: an additional state is required. Non-optimality of GPVW is shared by many of its descendants, e.g., [SB00].

Bound on excess length The following theorem establishes a bound on the excess length of counterexamples to some PLTLF formula ϕ resulting from an automaton that was constructed with the algorithm of [GPVW96]. The bound is linear in the future operator depth of ϕ.

Theorem 24 *Let ϕ be a PLTLF property and $B = B^{\neg\phi}_{GPVW}$ be a Büchi automaton constructed with GPVW [GPVW96]. Let $\alpha = \langle\beta, \gamma\rangle$ be a counterexample to ϕ. Then, there is an initialized fair path $\rho = \langle\sigma, \tau\rangle$ in B with $L(\rho) = \alpha$ and $|\sigma| \leq |\beta| + (h_f(\neg\phi) + 1)|\gamma|$ and $|\tau| = |\gamma|$.*

Proof: Assume $\alpha = \beta\gamma^\omega \in Lang(B)$. Hence, there exists a path ρ' in $\Pi^F(B)$ with $L(\rho') = \alpha$. Construct $\rho \in \Pi^F(B)$ as follows:

1. On the stem, just copy ρ': $\forall 0 \leq i < |\beta| \,.\, \rho[i] = \rho'[i]$.
2. On the loop, modify ρ' — while preserving acceptance of α — such that a $\mathbf{U}$-formula is fulfilled by satisfying its eventuality part as soon as possible. Since ϕ is a future time formula, this will always be the case within at most $|\gamma|$ steps. Similarly, if α permits to fulfil an $\mathbf{R}$-formula by making its eventuality part true, this is done and it is done as soon as possible. Finally, for each $\vee$-formula, choose the same expansion in each iteration of the loop. As a result, each subformula is expanded in the same way at a given position i in the loop in different iterations of the loop. Formally, we have:

*We assume algorithm and terminology as in [GPVW96].

(a) $\forall i \geq |\beta| \,.\, \forall \psi = \psi_1 \mathbf{U} \psi_2 \in \mathit{Old}(\rho[i]) \,.\, \exists 0 \leq j < |\gamma| \,.\, \psi_2 \in \mathit{Old}(\rho[i+j]) \wedge (\forall 0 \leq j' < j \,.\, \alpha, i+j' \not\models \psi_2) \wedge (\psi \in \mathit{Next}(\rho[i+j]) \Rightarrow \mathbf{X}\psi \in \mathit{Old}(\rho[i+j]))$

(b) $\forall \psi = \psi_1 \mathbf{R} \psi_2 \,.\, ((\exists i \geq |\beta| \,.\, \alpha, i \models \psi_1) \Rightarrow (\forall i \geq |\beta| \,.\, (\psi \in \mathit{Old}(\rho[i]) \Rightarrow (\exists 0 \leq j < |\gamma| \,.\, \psi_1 \in \mathit{Old}(\rho[i+j]) \wedge (\forall 0 \leq j' < j \,.\, \alpha, i+j' \not\models \psi_1) \wedge (\psi \in \mathit{Next}(\rho[i+j]) \Rightarrow \mathbf{X}\psi \in \mathit{Old}(\rho[i+j]))))))$

(c) $\forall i \geq |\beta| \,.\, \forall \psi = \psi_1 \vee \psi_2 \in \mathit{Old}(\rho[i]) \,.\, ((\psi_1 \notin \mathit{Old}(\rho[i]) \Rightarrow \forall k \,.\, (\psi \in \mathit{Old}(\rho[i+k|\gamma|]) \Rightarrow \psi_2 \in \mathit{Old}(\rho[i+k|\gamma|]))) \wedge (\psi_2 \notin \mathit{Old}(\rho[i]) \Rightarrow \forall k \,.\, (\psi \in \mathit{Old}(\rho[i+k|\gamma|]) \Rightarrow \psi_1 \in \mathit{Old}(\rho[i+k|\gamma|]))))$

It remains to show that ρ has the desired shape. We note the following (obvious) fact: From the construction of [GPVW96], each formula in $\mathit{Old}(\rho[i])$ is a subformula of one in $\mathit{New}(\rho[i])$ and each formula in $\mathit{Next}(\rho[i])$ is a subformula of one in $\mathit{Old}(\rho[i])$. Hence, each formula in some $\mathit{Old}(\rho[i+1])$ is a subformula of a formula in $\mathit{Old}(\rho[i])$: $\forall i \geq 0 \,.\, \forall \psi_1 \in \mathit{Old}(\rho[i+1]) \,.\, \exists \psi_2 \in \mathit{Old}(\rho[i]) \,.\, \psi_1 \in \mathit{sub}(\psi_2)$.

We now prove by induction that the Old-sets of ρ become stable after at most $h_f(\phi)+1$ loop iterations:

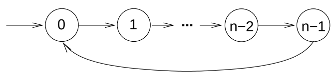
$$\begin{aligned}&\forall k \geq 0 \,.\, \forall 0 \leq i < |\gamma| \,.\, \forall k' > k \,.\\ &\quad \{\psi \mid \psi \in \mathit{Old}(\rho[|\beta|+k|\gamma|+i]) \wedge h_f(\psi) > h_f(\phi)-k\} =\\ &\quad \{\psi \mid \psi \in \mathit{Old}(\rho[|\beta|+k'|\gamma|+i]) \wedge h_f(\psi) > h_f(\phi)-k\}\end{aligned}$$

Base case. $k=0$: With the fact stated above, no formula in some Old-, Next-, or New-set can have future operator depth larger than ϕ. Hence, $\emptyset = \emptyset$.

Inductive case. Assume the claim holds for $k-1 \geq 0$.

"$\subseteq$" Let $\psi \in \mathit{Old}(\rho[|\beta|+k|\gamma|+i])$ with $h_f(\psi) = h_f(\phi)-k+1$. ψ is present in $\mathit{Old}(\rho[|\beta|+k|\gamma|+i])$ either because of an expansion of some $\psi' \in \mathit{Old}(\rho[|\beta|+k|\gamma|+i])$ with $\psi \in \mathit{sub}(\psi')$ or because ψ is contained in $\mathit{Next}(\rho[|\beta|+k|\gamma|+i-1])$.

1. ψ is present because of an expansion of $\psi' \in \mathit{Old}(\rho[|\beta|+k|\gamma|+i])$.
   (a) If ψ' is temporal, $h_f(\psi') > h_f(\psi)$. By i.a., ψ' will be present in $\mathit{Old}(\rho[|\beta|+k'|\gamma|+i])$ for all $k' > k$. By the construction of ρ above, ψ' will be expanded in the same way for each k' as for k. Hence, $\psi \in \mathit{Old}(\rho[|\beta|+k'|\gamma|+i])$.
   (b) If ψ' is Boolean, let ψ'' be the largest Boolean superformula of ψ' in $\mathit{sub}(\phi)$. ψ'' is present in $\mathit{Old}(\rho[|\beta|+k|\gamma|+i])$ either because of an expansion of a temporal formula ψ''' in $\mathit{Old}(\rho[|\beta|+k|\gamma|+i])$ or because $\psi''' = \mathbf{X}\psi'' \in \mathit{Old}(\rho[|\beta|+k|\gamma|+i-1])$. In both cases $h_f(\psi''') > h_f(\psi)$. I.a. and the construction of ρ guarantee the presence of ψ'' in $\mathit{Old}(\rho[|\beta|+k'|\gamma|+i])$ for all $k' > k$. The construction of ρ gives $\psi', \psi \in \mathit{Old}(\rho[|\beta|+k'|\gamma|+i])$.
2. ψ is contained in $\mathit{Next}(\rho[|\beta|+k|\gamma|+i-1])$.
   (a) If $\mathbf{X}\psi \in \mathit{Old}(\rho[|\beta|+k|\gamma|+i-1])$, i.a. and the construction of ρ prove $\psi \in \mathit{Old}(\rho[|\beta|+k'|\gamma|+i])$ for all $k' > k$.
   (b) Otherwise, ψ is a $\mathbf{U}$- or $\mathbf{R}$-formula. In that case, the same reasoning can be applied again. Because of the construction of ρ, at most $|\gamma|$ steps backward (i.e., 2 (b)) are required. Then either one of the cases 1 (a), 1 (b), or 2 (a) holds, a contradiction arises (a $\mathbf{U}$-formula must be fulfilled within $|\gamma|$



(a) model

$$
\begin{array}{l}
\neg\mathbf{G}((c \neq n-1)\ \mathbf{U}\ ((c = n-1) \wedge \\
\quad ((c \neq n-2)\ \mathbf{U}\ ((c = n-2) \wedge \\
\quad \ldots \\
\quad ((c \neq 0)\ \mathbf{U}\ (c = 0)) \\
\quad \ldots \\
\quad)) \\
\quad))
\end{array}
$$

(b) property for GPVW

$$
\begin{array}{l}
\neg(\mathbf{F}(\mathbf{G}(\mathbf{O}((c = 0) \wedge \\
\quad \mathbf{O}((c = 1) \wedge \\
\quad \ldots \\
\quad \mathbf{O}(c = n-1) \\
\quad \ldots \\
\quad) \\
\quad))))
\end{array}
$$

(c) property for KPR

Figure 5.3: Simple modulo-n counter with properties resulting in counterexamples of excess length linear in the future/past operator depth of the formulae.

steps), or ψ is a $\mathbf{R}$-formula, which is fulfilled by its right argument being continuously true.

"$\supseteq$" The proof is symmetrical to the "$\subseteq$"-case.

From the construction of [GPVW96] and the construction of ρ it is easy to see that stabilization of the Old-sets implies stabilization of the corresponding $Next$-sets:

$$
\begin{array}{l}
\forall k \geq 0\ .\ \forall 0 \leq i < |\gamma|\ .\ \forall k' > k\ . \\
\quad \{\psi \mid \psi \in Next(\rho[|\beta| + k|\gamma| + i]) \wedge h_f(\psi) > h_f(\phi) - k\} = \\
\quad \{\psi \mid \psi \in Next(\rho[|\beta| + k'|\gamma| + i]) \wedge h_f(\psi) > h_f(\phi) - k\}
\end{array}
$$

A state in B is uniquely determined by its Old- and $Next$-sets. It takes at most $h_f(\phi) + 1$ loop iterations before both, Old- and $Next$-sets, become stable. I.e., ρ has the desired shape. □

Approximate tightness of the bound The bound stated above is tight insofar as there is an example of a model and property such that the algorithm of Gerth et al. [GPVW96] produces a counterexample exhibiting excess length linear in the future operator depth of the formula (see Fig. 5.3 (a), (b)).

5.2.2 Kesten et al. (KPR)

Tightness for PLTLF In a Büchi automaton B^{ϕ}_{KPR} generated by the algorithm of Kesten et al. [KPR98] each state variable corresponds to a subformula ψ of ϕ (see Tab. 2.2). This directly proves tightness of B^{ϕ}_{KPR} for a PLTLF formula ϕ:

Theorem 25 *Let ϕ be a future time LTL formula, let B^{ϕ}_{KPR} be defined as in Sect. 2.6. Then B^{ϕ}_{KPR} is tight.*

Proof: Every two states in $B = B^{\phi}_{KPR}$ differ in the valuation of at least one state variable, and therefore specify a different, non-overlapping future (which includes presence). According to Thm. 20, a Büchi automaton B is tight iff for each accepted word α there exists a path ρ in $\Pi^F(B)$ with $L(\rho) = \alpha$ and $\forall i, j\ .\ (\alpha[i, \infty] = \alpha[j, \infty] \Rightarrow \rho[i] = \rho[j])$. Clearly, $\alpha[i, \infty] = \alpha[j, \infty]$ have the same future, hence, on each run in B we have $\alpha[i, \infty] = \alpha[j, \infty] \Rightarrow \rho[i] = \rho(j)$. □

Bound on excess length What is useful for future time hurts tightness when past operators are included: B^{ϕ}_{KPR} also distinguishes states of an accepted word that have different past but same future. Lemma 4 states that a past time formula can distinguish only finitely many iterations of a loop. This can be used to establish an upper bound on the excess length of a counterexample to a PLTLB formula obtained from a Büchi automaton that was constructed with KPR [KPR98]:

Theorem 26 *Let ϕ be a PLTLB property and $B = B^{\neg\phi}_{KPR}$ a Büchi automaton constructed with KPR [KPR98]. Let $\alpha = \langle \beta, \gamma \rangle$ be a counterexample to ϕ. Then, there is an initialized fair path $\rho = \langle \sigma, \tau \rangle$ in B with $L(\rho) = \alpha$ and $|\sigma| \leq |\beta| + (h(\neg\phi) + 1)|\gamma|$ and $|\tau| = |\gamma|$.*

Proof: The states of B each correspond to a subset of $\{\psi \mid \psi \in sub(\neg\phi)\} \cup \{\circ\psi \mid \circ \in \{\mathbf{X}, \mathbf{Y}\} \wedge \psi \in sub(\neg\phi)\}$. By Lemma 4, a PLTLB formula cannot distinguish iterations of the loop that occur after the $h(\neg\phi)$-th iteration. More formally, for any lasso $\alpha = \langle \beta, \gamma \rangle$, any PLTLB formula ψ, and any $i \geq |\beta| + h(\psi)|\gamma|$, $\alpha[i, \infty] \models \psi$ iff $\alpha[i + |\gamma|, \infty] \models \psi$. Hence, by the correctness of the construction, one can derive an initialized fair path ρ in B from α. By Lemma 4, a loop of length γ starts in ρ after at most $h(\neg\phi) + 1$ iterations of the loop in α have passed (note that the past time operator depth of the formulae labelling the states of B may be $h(\neg\phi) + 1$). □

Approximate tightness of the bound For an example that exhibits excess length, which is linear in the past operator depth of the formula, consider the simple modulo-n counter and property in Fig. 5.3 (a), (c) (adapted from [BC03]). The innermost formula $\mathbf{O}(c = n - 1)$ remains true from the end of the first loop iteration in the counter, $\mathbf{O}((c = n - 2) \wedge (\mathbf{O}(c = n - 1)))$ becomes and remains true $n - 1$ steps later, etc. Hence, a loop in $B^{\neg\phi}_{KPR}$ is only reached after $\mathbf{O}(n^2)$ steps of the counter have been performed. Clearly, the shortest counterexample is a single iteration of the loop with $\mathbf{O}(n)$ steps.

A (too) costly solution Every PLTLB formula can be transformed into a future time LTL formula equivalent at the beginning of a sequence [Gab89]. Due to [LMS02] we can expect an at least exponential worst-case increase in the size of the formula. Rather than translating an LTL formula with past into a pure future version, we follow a different path in the next section.

5.3 A Tight Look at LTL Model Checking

Theorem 26 states that a Büchi automaton constructed with KPR [KPR98] accepts a shortest counterexample with a path that may have an overly long stem but a loop of the same length as that of the counterexample. Bounded model checking [BCCZ99] has been extended recently to include past time operators [BC03, CRS04, LBHJ05]. Of these, [BC03, LBHJ05] use *virtual unrolling* of the transition relation to find shortest counterexamples if past time operators are present. Inspired by [LBHJ05], we adapt this approach to construct a tight Büchi automaton for PLTLB based on KPR [KPR98].

5.3.1 Virtual Unrolling for Bounded Model Checking of PLTLB

Encoding BMC for PLTLF In bounded model checking, typically one fresh Boolean variable $x_{i,\psi}$ is introduced for each pair of relative position in the path ($0 \leq i \leq k$) and subformula

ψ of ϕ, such that $x_{i,\psi}$ is true iff ψ holds at position i. On a lasso-shaped path, the truth of a future time formula ϕ at position i may depend on the truth of some of its subformulae ψ at positions $i' > i$. While those are not available directly, the truth of a future time formula at a given position within the loop does not change between different iterations of the loop. Hence, the truth value of ψ at position $0 \leq i < k-l$ in any iteration $m \geq 0$ of the loop can be substituted with the truth value of ψ at position i in the first iteration: $\rho[l+m(k-l)+i, \infty] \models \psi \Leftrightarrow \rho[l+i, \infty] \models \psi$. A single unrolling of the loop is therefore sufficient, resulting in a shortest counterexample.

The problem with PLTLB When past time operators are admitted, this is no longer true. By Lemma 4, the truth of a subformula ψ may change between the first $h_p(\psi)$ iterations of the loop before it reaches a stable value at iteration $h_p(\psi)+1$. Hence, only after $h_p(\psi)+1$ iterations can the truth value of ψ in some iteration $m > h_p(\psi)+1$ of the loop be replaced by the truth value of ψ in iteration $h_p(\psi)+1$: $\rho[l+m(k-l)+i, \infty] \models \psi \Leftrightarrow \rho[l+(h_p(\psi)+1)(k-l)+i, \infty] \models \psi$. A naive approach for checking a past time formula ϕ would still have one Boolean variable per pair of relative position in the path and subformula. However, the approach would have to ensure that the path ends with $h_p(\phi)+1$ copies of the loop. This would lead to a more complicated formulation of loop detection and would not allow to find shortest counterexamples. A less naive, but still suboptimal solution might not guarantee a high enough number of loop unrollings directly, but could include the variables representing the truth of properties in the loop detection. That approach could not ensure shortest counterexamples either.

Solution Benedetti and Cimatti [BC03] showed how to do better: note, that some subformulae ψ of ϕ have lower past operator depth, and, therefore, require fewer loop iterations to stabilize. In particular, atomic propositions remain stable from the first iteration onward. It is sufficient to perform a single unrolling of the loop. Rather than having only one Boolean variable $x_{i,\psi}$ per pair of relative position i in the path and subformula ψ, there are now as many variables per pair (i, ψ) as iterations of the loop are required for that subformula to stabilize. Each variable corresponds to the truth value of ψ at the same relative position i but in a different iteration m of the loop: $x_{i,\psi,m} \Leftrightarrow \rho[i+m(k-l), \infty] \models \psi$ with $0 \leq i \leq k \wedge 0 \leq m \leq h_p(\psi)$ (the value of $x_{i,\psi,m}$ may not be well-defined if $m > 0 \wedge i < l$). This *virtual unrolling* of the loop leads to shortest counterexamples.

5.3.2 A Tight Büchi Automaton for PLTLB

Same problem, same solution A Büchi automaton constructed with KPR [KPR98] suffers from similar problems as the naive approaches to bounded model checking of PLTLB. The automaton has a single variable representing the truth of a subformula in a given state. For a loop in the product of the model and the automaton to occur, the truth of all subformulae must have stabilized. Hence, we adopt the same idea as outlined above to obtain a tight Büchi automaton.

Definition The following definition formally states the construction of a tight Büchi automaton for PLTLB.

Definition 7 *We symbolically construct a Büchi automaton* $B^{\phi}_{SB} = (V^{\phi}_{SB}, S^{\phi}_{SB}, T^{\phi}_{SB}, I^{\phi}_{SB}, L^{\phi}_{SB}, F^{\phi}_{SB})$ *for a PLTLB formula* ϕ *as follows.* $AP^{\phi} = \{p \mid p \text{ is an atomic proposition in } sub(\phi)\}$*,* $V^{\phi}_{SB} = V^{\phi} \cup \{lo\}$*, where all state variables in* V^{ϕ} *are Boolean and* lo *has range* $\{st, lb, le\}$*,* $S^{\phi}_{SB} = S^{\phi}$*,* $T^{\phi}_{SB} = T^{\phi} \wedge (lo \neq st \rightarrow lo' \neq st)$*,* $I^{\phi}_{SB} = I^{\phi} \wedge x_{\phi,0}$*,* $F^{\phi}_{SB} = F^{\phi} \cup \{lo = le\}$*, and* $L^{\phi}_{SB}(s) = \{p \mid x_{p,0}(s) = 1\}$*.* V^{ϕ}*,* S^{ϕ}*,* T^{ϕ}*,* I^{ϕ}*, and* F^{ϕ} *are defined recursively in Tab. 5.1.*

Each subformula ψ of ϕ is represented by $h_p(\psi) + 1$ state variables $x_{\psi,m}$. We refer to the m in $x_{\psi,m}$ as *generation* below. One more state variable lo (for *l*ass*o*, see also Sect. 3.1) with values *st*em, *l*oop *b*ody, and *l*oop *e*nd is added. As long as lo has value st (on the stem), only variables in generation 0 are constrained according to the recursive definition of PLTLB. When lo becomes lb (on the loop), the definitions apply to all generations. While $lo = lb$ (the end of a loop iteration is not yet reached), $x_{\psi,m}$ is defined in terms of current and next-state values of variables in the same generation. When $lo = le$ (at the end of a loop iteration), the next-state values are obtained from the next generation of variables if the present generation is not already the last. The fairness constraints, which guarantee the correct fixed point for $\mathbf{U}$ formulae, are only applied to the last generation of the corresponding variables.

The intuition is as follows. Starting with generation 0 on the stem and the first iteration of the loop, each generation m of $x_{\psi,m}$ represents the truth of ψ in one loop iteration, the end of which is signaled by $lo = le$. Formally, for $m < h_p(\psi)$, $x_{\psi,m}(i)$ holds the truth of ψ at position i of a word iff lo has had value le for m times prior to the current state. From the $h_p(\psi)$-th occurrence of $lo = le$, $x_{\psi,h_p(\psi)}$ continues to represent the truth of ψ.

Note that lo is an oracle.[†] The valuation of this variable on an arbitrary run may not correspond to the situation it is named after. However, for B^{ϕ}_{SB} to correctly recognize $\{\alpha \mid \alpha \models \phi\}$, it is not relevant which generation holds the truth at a given position. It is only required that at each position some generation represents truth correctly, each generation passes on to the next at some point, and ultimately, depending on ψ, the last generation $h_p(\psi)$ continues to hold the proper values.

For tightness, the variables of a given generation need to be able to take on the same values in every iteration of the loop, regardless of whether they currently hold the truth or not. This requires breaking the links to previous iterations for variables of generation 0 representing $\mathbf{Y}$ and $\mathbf{S}$ formulae at each start of a loop iteration after the first. In addition, $\mathbf{Y}$- and $\mathbf{S}$-variables of generations > 0 may not be constrained by past values at the beginning of the loop body. On a shortest run on some lasso-shaped word α, lo will correctly signal loop body and loop end.

Correctness, completeness, tightness In the following we establish that the language of B^{ϕ}_{SB} is indeed ϕ and that B^{ϕ}_{SB} is tight.

Theorem 27 *Let* ϕ *be a PLTLB formula, let* B^{ϕ}_{SB} *be defined as above. Then,* $Lang(B^{\phi}_{SB}) = \{\alpha \mid \alpha \models \phi\}$ *and* B^{ϕ}_{SB} *is tight.*

Proof: By Lemma 28 and 29. □

Lemma 28 $Lang(B^{\phi}_{SB}) = \{\alpha \mid \alpha \models \phi\}$

[†] If the automaton is combined with the state-recording translation, lo can be provided by the translation, see also Sect. 7.2.

ψ	definition
p	$V^\psi = \{x_{p,0}\}$ $S^\psi = x_{p,0} \leftrightarrow p$ $T^\psi = 1$ $I^\psi = 1$ $F^\psi = \emptyset$
$\neg\psi_1$	$V^\psi = V^{\psi_1} \cup \bigcup_{m=0}^{h_p(\psi)} \{x_{\psi,m}\}$ $S^\psi = S^{\psi_1} \wedge \bigwedge_{m=0}^{h_p(\psi)} (x_{\psi,m} \leftrightarrow \neg x_{\psi_1,m})$ $T^\psi = 1$ $I^\psi = I^{\psi_1}$ $F^\psi = F^{\psi_1}$
$\psi_1 \vee \psi_2$	$V^\psi = V^{\psi_1} \cup V^{\psi_2} \cup \bigcup_{m=0}^{h_p(\psi)} \{x_{\psi,m}\}$ $S^\psi = S^{\psi_1} \wedge S^{\psi_2} \wedge \bigwedge_{m=0}^{h_p(\psi)} (x_{\psi,m} \leftrightarrow x_{\psi_1,\min(m,h_p(\psi_1))} \vee x_{\psi_2,\min(m,h_p(\psi_2))})$ $T^\psi = T^{\psi_1} \wedge T^{\psi_2}$ $I^\psi = I^{\psi_1} \wedge I^{\psi_2}$ $F^\psi = F^{\psi_1} \cup F^{\psi_2}$
$\mathbf{X}\psi_1$	$V^\psi = V^{\psi_1} \cup \bigcup_{m=0}^{h_p(\psi)} \{x_{\psi,m}\}$ $S^\psi = S^{\psi_1}$ $T^\psi = T^{\psi_1} \wedge (\mathit{lo} = \mathit{st} \rightarrow (x_{\psi,0} \leftrightarrow x'_{\psi_1,0}))$ $\wedge (\mathit{lo} = \mathit{lb} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-1} (x_{\psi,m} \leftrightarrow x'_{\psi_1,m}))$ $\wedge (\mathit{lo} = \mathit{le} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-1} (x_{\psi,m} \leftrightarrow x'_{\psi_1,m+1}))$ $\wedge (\mathit{lo} \neq \mathit{st} \rightarrow (x_{\psi,h_p(\psi)} \leftrightarrow x'_{\psi_1,h_p(\psi_1)}))$ $I^\psi = I^{\psi_1}$ $F^\psi = F^{\psi_1}$
$\psi_1 \ \mathbf{U}\ \psi_2$	$V^\psi = V^{\psi_1} \cup V^{\psi_2} \cup \bigcup_{m=0}^{h_p(\psi)} \{x_{\psi,m}\}$ $S^\psi = S^{\psi_1} \wedge S^{\psi_2}$ $T^\psi = T^{\psi_1} \wedge T^{\psi_2} \wedge (\mathit{lo} = \mathit{st} \rightarrow (x_{\psi,0} \leftrightarrow x_{\psi_2,0} \vee (x_{\psi_1,0} \wedge x'_{\psi,0})))$ $\wedge (\mathit{lo} = \mathit{lb} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-1} (x_{\psi,m} \leftrightarrow x_{\psi_2,\min(m,h_p(\psi_2))} \vee (x_{\psi_1,\min(m,h_p(\psi_1))} \wedge x'_{\psi,m})))$ $\wedge (\mathit{lo} = \mathit{le} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-1} (x_{\psi,m} \leftrightarrow x_{\psi_2,\min(m,h_p(\psi_2))} \vee (x_{\psi_1,\min(m,h_p(\psi_1))} \wedge x'_{\psi,m+1})))$ $\wedge (\mathit{lo} \neq \mathit{st} \rightarrow (x_{\psi,h_p(\psi)} \leftrightarrow x_{\psi_2,h_p(\psi_2)} \vee (x_{\psi_1,h_p(\psi_1)} \wedge x'_{\psi,h_p(\psi)})))$ $I^\psi = I^{\psi_1} \wedge I^{\psi_2}$ $F^\psi = F^{\psi_1} \cup F^{\psi_2} \cup \{\{\neg x_{\psi,h_p(\psi)} \vee x_{\psi_2,h_p(\psi_2)}\}\}$
$\mathbf{Y}\psi_1$	$V^\psi = V^{\psi_1} \cup \bigcup_{m=0}^{h_p(\psi)} \{x_{\psi,m}\}$ $S^\psi = S^{\psi_1}$ $T^\psi = T^{\psi_1} \wedge (\mathit{lo} = \mathit{st} \rightarrow (x'_{\psi,0} \leftrightarrow x_{\psi_1,0}))$ $\wedge (\mathit{lo} = \mathit{lb} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-1} (x'_{\psi,m} \leftrightarrow x_{\psi_1,m}))$ $\wedge (\mathit{lo} = \mathit{le} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-2} (x'_{\psi,m+1} \leftrightarrow x_{\psi_1,m}))$ $\wedge (\mathit{lo} \neq \mathit{st} \rightarrow (x'_{\psi,h_p(\psi)} \leftrightarrow x_{\psi_1,h_p(\psi_1)}))$ $I^\psi = I^{\psi_1} \wedge (x_{\psi,0} \leftrightarrow 0)$ $F^\psi = F^{\psi_1}$
$\psi_1 \ \mathbf{S}\ \psi_2$	$V^\psi = V^{\psi_1} \cup V^{\psi_2} \cup \bigcup_{m=0}^{h_p(\psi)} \{x_{\psi,m}\}$ $S^\psi = S^{\psi_1} \wedge S^{\psi_2}$ $T^\psi = T^{\psi_1} \wedge T^{\psi_2} \wedge (\mathit{lo} = \mathit{st} \rightarrow (x'_{\psi,0} \leftrightarrow x'_{\psi_2,0} \vee (x'_{\psi_1,0} \wedge x_{\psi,0})))$ $\wedge (\mathit{lo} = \mathit{lb} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-1} (x'_{\psi,m} \leftrightarrow x'_{\psi_2,\min(m,h_p(\psi_2))} \vee (x'_{\psi_1,\min(m,h_p(\psi_1))} \wedge x_{\psi,m})))$ $\wedge (\mathit{lo} = \mathit{le} \rightarrow \bigwedge_{m=0}^{h_p(\psi)-1} (x'_{\psi,m+1} \leftrightarrow x'_{\psi_2,\min(m+1,h_p(\psi_2))} \vee (x'_{\psi_1,\min(m+1,h_p(\psi_1))} \wedge x_{\psi,m})))$ $\wedge (\mathit{lo} \neq \mathit{st} \rightarrow (x'_{\psi,h_p(\psi)} \leftrightarrow x'_{\psi_2,h_p(\psi_2)} \vee (x'_{\psi_1,h_p(\psi_1)} \wedge x_{\psi,h_p(\psi)})))$ $I^\psi = I^{\psi_1} \wedge I^{\psi_2} \wedge (x_{\psi,0} \leftrightarrow x_{\psi_2,0})$ $F^\psi = F^{\psi_1} \cup F^{\psi_2}$

Table 5.1: Property-dependent part of a tight Büchi automaton

Proof: (Correctness) We show that on every initialized fair path in $(V^{\phi}_{SB}, S^{\phi}_{SB}, T^{\phi}_{SB}, I^{\phi}, L^{\phi}_{SB}, F^{\phi}_{SB})$ the values of $x_{\psi,m_i}(i)$ represent the validity of the subformula ψ at position i, where m_i is either the number of $(lo = le)$'s seen so far or $h_p(\psi)$, whichever is smaller. Formally, let ρ be an initialized fair path with $L^{\phi}_{SB}(\rho) = \alpha$ in $(V^{\phi}_{SB}, S^{\phi}_{SB}, T^{\phi}_{SB}, I^{\phi}, L^{\phi}_{SB}, F^{\phi}_{SB})$. For each position i in α, let $m_i = \min(|\{j \mid (j \leq i-1) \wedge lo(\rho(j)) = le\}|, h_p(\psi))$. Inspection of Tab. 5.1 shows that the constraints on the $x_{\psi,m_i}(i)$ are the same as the constraints on the corresponding $x_{\psi}(i)$ in Tab. 2.2. Hence, $\alpha[i,\infty] \models \psi \Leftrightarrow x_{\psi,m_i}(\rho(i))$.

(Completeness) We show that there is an initialized fair path ρ in $(V^{\phi}_{SB}, S^{\phi}_{SB}, T^{\phi}_{SB}, I^{\phi}, L^{\phi}_{SB}, F^{\phi}_{SB})$ with $L^{\phi}_{SB}(\rho) = \alpha$ for each word α. Choose a set of indices $U = \{i_0, i_1, \ldots\}$ (for "up") such that $lo(\rho(i)) = le \leftrightarrow i \in U$. Further, choose $ls \leq i_0$ and set $lo(\rho(j)) = st \leftrightarrow j < ls$. We inductively construct a valuation for $x_{\psi,m}(i)$ for each subformula ψ of ϕ, $m \leq h_p(\psi)$, and $i \geq 0$. If ψ is an atomic proposition p, set $x_{p,0}(i) \leftrightarrow (\alpha[i,\infty] \models p)$. If the top level operator of ψ is Boolean, the valuation follows directly from the semantics of the operator. For $\mathbf{X}$, each $x_{\psi,m}(i)$ is defined at most once in Tab. 5.1. $\psi = \mathbf{Y}\psi_1$ is similar. Note that $h_p(\psi) = h_p(\psi_1) + 1$. Therefore, m runs only up to $h_p(\psi) - 2$ if $lo = le$; $m = h_p(\psi) - 1$ is covered by the case for $lo \neq st$ in the line below. $x_{\psi,m}(i)$ is unconstrained if $m = 0$ and $i - 1 \in U$ as well as if $m \geq 1$ and $i \leq ls$. For $\psi = \psi_1 \ \mathbf{U} \ \psi_2$, start with generation $h_p(\psi)$. If $x_{\psi_2,h_p(\psi_2)}$ remains false from some i_m on, assign $\forall i \geq i_m \ . \ x_{\psi,h_p(\psi)}(i) \leftrightarrow 0$. Now work towards decreasing i from each i_n with $x_{\psi_2,h_p(\psi_2)}(i_n) \leftrightarrow 1$, using line 4 in the definition of T for $\mathbf{U}$. Continue with generation $h_p(\psi) - 1$. Start at each $i \in U$ by obtaining $x_{\psi,h_p(\psi)-1}(i)$ from the previously assigned $x_{\psi,h_p(\psi)}(i+1)$ via line 3. Then work towards decreasing i again, using lines 1 or 2 in the definition of T until $x_{\psi,h_p(\psi)-1}$ is assigned for all i. This is repeated in decreasing order for each generation $0 \leq m < h_p(\psi) - 1$. For $\mathbf{S}$, start with $x_{\psi,0}(0)$ and proceed towards increasing i, also increasing m when $i \in U$ (lines 1 – 3 in the definition of T for $\mathbf{S}$). When $m = h_p(\psi)$ is reached, assign $x_{\psi,h_p(\psi)}(i)$ for all i using the fourth line in the definition of T. Then, similar to $\mathbf{U}$, work towards decreasing m and i from each $i \in U$. Fairness follows from the definition of U, ls, and the valuation chosen for $\mathbf{U}$.

The claim is now immediate by the definition of I^{ϕ}_{SB}. □

Lemma 29 *B^{ϕ}_{SB} is tight.*

Proof: We show inductively that the valuations of the variables $x_{\psi,m}(i)$ can be chosen such that the valuation at a given relative position in a loop iteration is the same for each iteration in a generation m. Formally, let $\alpha = \beta\gamma^{\omega}$ with $\alpha \models \phi$. There exists a run ρ on α such that for all subformulae ψ of ϕ

$$\forall m \leq h_p(\psi) \ . \ \forall i_1, i_2 \geq |\beta| \ . \ ((\exists k \geq 0 \ . \ i_2 - i_1 = k|\gamma|) \Rightarrow (x_{\psi,m}(\rho(i_1)) \leftrightarrow x_{\psi,m}(\rho(i_2))))$$

Atomic propositions, Boolean connectives, and $\mathbf{X}$ are clear. $\mathbf{Y}$ is also easy, we only have to assign the appropriate value from other iterations when $x_{\psi,m}(i)$ is unconstrained. For $\psi = \psi_1 \ \mathbf{U} \ \psi_2$, by the induction hypothesis, $x_{\psi_2,h_p(\psi_2)}$ is either always false (in which case we assigned $x_{\psi,h_p(\psi)}(i)$ to false according to the proof of Lemma 28) or becomes true at the same time in each loop iteration. Hence, the claim holds for generation $h_p(\psi)$. From there we can proceed to previous generations in the same manner as in the proof of Lemma 28. For $\mathbf{S}$ we follow the order of assignments from the proof of Lemma 28. By induction, the claim holds for generation

$h_p(\psi)$. From there, we proceed towards decreasing i and m. We use, by induction, the same valuations of subformulae and the same equations (though in reverse direction) as we used to get from $x_{\psi,0}(0)$ to generation $h_p(\psi)$. □

Complexity We immediately have the following corollary. Note, that the size of a Büchi automaton that is tight in the original sense of [KV01] (i.e., it recognizes shortest violating prefixes of safety properties) is doubly exponential in $|\phi|$ [KV01].

Corollary 30 *Let ϕ be a PLTLB formula. There is a tight Büchi automaton B with $Lang(B) = \phi$ with $\mathrm{O}(2^{|\phi|^2})$ states. A symbolic representation of B can be constructed in $\mathrm{O}(|\phi|^2)$ time and space.*

Remarks The same optimization as mentioned in Sect. 2.2 for KPR [KPR98] can be applied. It replaces state variables for Boolean connectives with macros in order to reduce the number of BDD variables in the context of symbolic model checking with BDDs. Note also, that for a PLTLF formula ϕ, B^{ϕ}_{SB} mostly reduces to B^{ϕ}_{KPR}: only one state variable $x_{\psi,0}$ is introduced per formula, the remaining implications $lo = st$ and $lo \neq st$ have the same right hand sides, and fairness is defined on generation 0. Finally, remember that bit-set degeneralization must be used rather than Choueka's flag construction to preserve tightness after degeneralization (see also Sect. 2.3).

5.3.3 Partial Unrolling

A similar optimization can be applied to the tight Büchi automaton as sketched in [HJL05]: virtual unrolling need not be performed to the full past operator depth of the formula, but can be adjusted according to the user's needs. In fact, if $h_p(\psi)$ in Def. 7 is replaced with $h'_p(\psi) = \min(h_p(\psi), n)$ for some $n \geq 0$, the construction above still yields an automaton $B^{\phi}_{SB,n}$ such that $Lang(B^{\phi}_{SB,n}) = \{\alpha \mid \alpha \models \phi\}$. The case $n = 0$ leads to an automaton that is, apart from the oracle lo, very similar to [KPR98]. The case $n \geq h_p(\phi)$ gives a tight automaton. While $B^{\phi}_{SB,n}$ may not be tight for $n < h_p(\phi)$, it is smaller than B^{ϕ}_{SB}, and, therefore, it may allow the user to trade performance for counterexample length.

5.4 Generalization

Definition The construction in the previous section deals only with a too long stem σ. In Def. 8 we show how to generalize the construction of a tight Büchi automaton for a PLTLB formula to obtain a tight Büchi automaton B_t from an arbitrary Büchi automaton B. Let $\alpha = \beta\gamma^\omega$ be a lasso shaped counterexample, let $\rho = \sigma\tau^\omega$ be a run on α. ρ may have both, a too long stem σ (i.e., σ continues on γ^ω), and a loop τ such that $lcm(|\tau|, |\gamma|) \neq |\gamma|$. To fit $\rho = \sigma\tau^\omega$ in the shape of $\alpha = \beta\gamma^\omega$ we form a run $\rho_t = \sigma_t\tau_t^\omega$ with $|\sigma_t| = |\beta|$ and $|\tau_t| = |\gamma|$. The states of ρ_t are vectors of states of ρ. The construction of σ_t is easy, it basically just copies the first $|\beta|$ states from ρ. To obtain τ_t we "wind up" in zig-zag manner potentially remaining states from σ and enough repetition of τ to form a loop of vectors. The role of the oracle lo, as in the previous section, is to indicate loop start and end so that the parallel parts of the original run

Definition 8 *Let $B = (S, T, I, L, F)$ be a generalized Büchi automaton. Then B_t is defined as $B_t = (S_t, T_t, I_t, L_t, F_t)$ with*

$$
\begin{array}{rcl}
S_t & = & \bigcup_{m=0}^{|S|} \bigcup_{n=1}^{|F||S|} \{(s_1, \ldots, s_m, (l_1, f_{1,1} \ldots, f_{1,|F|}), \ldots, (l_n, f_{n,1} \ldots, f_{n,|F|}), lo)\} \\
& & \textit{where} \\
& & s_1, \ldots, s_m, l_1, \ldots, l_n \in S \wedge \\
& & L(s_1) = \ldots = L(s_m) = L(l_1) = \ldots = L(l_n) \wedge \\
& & f_{1,1}, \ldots, f_{n,|F|} \in \mathbb{B} \wedge \\
& & lo \in \{st, lb, le\} \\
T_t & = & \{((s_1, \ldots, s_m, (l_1, f_{1,1}, \ldots, f_{1,|F|}), \ldots, (l_n, f_{n,1}, \ldots, f_{n,|F|}), lo), \\
& & \quad (s'_1, \ldots, s'_m, (l'_1, f'_{1,1}, \ldots, f'_{1,|F|}), \ldots, (l'_n, f'_{n,1}, \ldots, f'_{n,|F|}), lo')) \mid \\
& & \quad (lo = st \rightarrow (s_1, s'_1) \in T) \wedge \\
& & \quad (lo = lb \rightarrow \bigwedge_{p=1}^{m} (s_p, s'_p) \in T \wedge \\
& & \qquad \bigwedge_{q=1}^{n} (l_q, l'_q) \in T \wedge \\
& & \qquad \bigwedge_{q=1}^{n} \bigwedge_{k=1}^{|F|} (f'_{q,k} \rightarrow l'_q \in F_k \vee f_{q,k})) \wedge \\
& & \quad (lo = le \rightarrow (\bigwedge_{p=1}^{m-1} (s_p, s'_{p+1}) \in T) \wedge (s_m, l'_1) \in T \wedge \\
& & \qquad (\bigwedge_{q=1}^{n-1} (l_q, l'_{q+1}) \in T) \wedge (l_n, l'_1) \in T \wedge \\
& & \qquad \bigwedge_{k=1}^{|F|} (f'_{1,k} \rightarrow l'_1 \in F_k) \wedge \\
& & \qquad \bigwedge_{q=1}^{n-1} \bigwedge_{k=1}^{|F|} (f'_{q+1,k} \rightarrow l'_{q+1} \in F_k \vee f_{q,k}) \wedge \\
& & \qquad \bigwedge_{k=1}^{|F|} f_{n,k}) \wedge \\
& & \quad (lo \neq st \rightarrow lo' \neq st)\} \\
I_t & = & \{(s_1 \ldots, s_m, (l_1, f_{1,1}, \ldots, f_{1,|F|}), \ldots, (l_n, f_{n,1}, \ldots, f_{n,|F|}), lo) \mid \\
& & \quad (m > 0 \rightarrow s_1 \in I) \wedge (m = 0 \rightarrow l_1 \in I \wedge lo \neq st)\} \\
L_t & = & (s_1 \ldots, s_m, (l_1, f_{1,1}, \ldots, f_{1,|F|}), \ldots, (l_n, f_{n,1}, \ldots, f_{n,|F|}), lo) \\
& & \mapsto L(s_1) = \ldots = L(s_m) = L(l_1) = \ldots = L(l_n) \\
F_t & = & \{\{(s_1 \ldots, s_m, (l_1, f_{1,1}, \ldots, f_{1,|F|}), \ldots, (l_n, f_{n,1}, \ldots, f_{n,|F|}), le)\}\}
\end{array}
$$

can be connected accordingly. In effect, several parts of the original run on α in B now run in parallel in B_t.

Correctness, completeness, tightness Below we prove that the construction yields a tight automaton that accepts the same language as the original automaton.

Lemma 31 *$Lang(B) = Lang(B_t)$.*

Proof: "$\subseteq$": Let ρ be a run on α in B. Technically, we construct a run ρ_t in B_t, which need not accept α in a shortest way, by embedding ρ in B_t. We set $m = 0, n = 1$:

$$\forall i \geq 0 \,.\, \rho_t[i] = ((\rho[i], f_{1,1}[i], \ldots, f_{1,|F|}[i]), lo[i])$$

By definition of a run ρ is fair. Hence, there are infinite sequences of indices ξ_k of states in ρ for each F_k such that

1. ξ_k contains only indices of fair states: $\forall j \geq 0 \,.\, \rho[\xi_k[j]] \in F_k$; and

2. (note: "position" $\equiv$ index in ξ) all indices at position $j+1$ of ξ_k are larger than the largest index at position j of $\xi_{k'}$ for any k': $\forall j \geq 0 \,.\, \forall 1 \leq k, k' \leq |F| \,.\, \xi_k[j] < \xi_{k'}[j+1]$.

We define ξ_{max} as the sequence of the maximal indices in ξ_k: $\forall j \geq 0 \,.\, \xi_{max}[j] = max_{k=1}^{|F|} \xi_k[j]$. ξ_{max} gives us a sequence of intervals in ρ such that all fairness constraints are fulfilled in any such interval. We define a recursive function iv that relates an index in ρ to its interval:

$$\begin{array}{lcl} iv(0) & = & 0 \\ iv(i+1) & = & \left\{ \begin{array}{ll} iv(i) & \text{if} \quad i \neq \xi_{max}[iv(i)] \\ iv(i)+1 & \text{otherwise} \end{array} \right. \end{array}$$

Now we can finally set

$$\begin{array}{lcl} f_{1,k}[i] & \leftrightarrow & i \geq \xi_k(iv(i)) \\ lo[i] = lb & \leftrightarrow & i \neq \xi_{max}(iv(i)) \\ lo[i] = le & \leftrightarrow & i = \xi_{max}(iv(i)) \end{array}$$

"$\supseteq$": Let ρ_t be a run on α in B_t. We extract a run ρ on α in B by selecting one (component-) state from each from each (vector-) state in ρ_t. Let $gen(i)$ be defined as follows:

$$\begin{array}{lcl} gen(0) & = & 1 \\ gen(i+1) & = & \left\{ \begin{array}{lll} gen(i) & \text{if} & lo(\rho_t[i]) \neq le \\ gen(i)+1 & \text{otherwise, if} & lo(\rho_t[i]) = le \ \wedge \ gen(i) < m+n \\ m+1 & \text{otherwise, if} & lo(\rho_t[i]) = le \ \wedge \ gen(i) = m+n \end{array} \right. \end{array}$$

With that, ρ, defined by

$$\rho[i] \quad = \quad \left\{ \begin{array}{ll} s_{gen(i)}(\rho_t[i]) & \text{if} \quad gen(i) \leq m \\ l_{gen(i)-m}(\rho_t[i]) & \text{otherwise} \end{array} \right.$$

is a run on α in B: $\rho[0] = \left\{ \begin{array}{ll} s_1(\rho_t[0]) & \text{if} \quad m > 0 \\ l_1(\rho_t[0]) & \text{otherwise} \end{array} \right.$ is, by definition of I_t, an initial state in B. Further, by definition of T_t, $(\rho[i], \rho[i+1]) \in T$. As ρ_t is fair, lo has value le infinitely often, and therefore $gen(i) = m+n$ and $gen(i+1) = m+1$ infinitely often. Fairness of ρ follows then directly from the definition of T_t. Finally, $L(\rho_t[i]) = L(\rho[i])$. □

Lemma 32 *B_t is tight.*

Proof: Let $\rho = \sigma\tau^\omega$ be a run in B on a counterexample $\alpha = \beta\gamma^\omega$. Assume that $|\sigma| \geq |\beta|$ (otherwise, extend σ with as many τ's as necessary). The "excess part" of σ, $\sigma[|\beta|, |\sigma| - 1]$, forms tuples with characters from γ. Note that there are at most $|S|$ different states of B available for each position of $|\gamma|$ to form a tuple. We therefore assume that $|\sigma[|\beta|, |\gamma| - 1]| \leq |S||\gamma|$. Otherwise, there are $0 \leq l_1 < l_2 \leq |S|$ such that $\rho[|\beta| + l_1|\gamma|] = \rho[|\beta| + l_2|\gamma|]$, and σ can be shortened accordingly. For a similar bound on the length of a combined loop in α and ρ, consider that a fair loop in B might take at most $|F||S|$ steps, hence, the combined loop takes at most $|F||S||\gamma|$ steps.

We construct a run $\rho_t = \sigma_t{\tau_t}^\omega$ on α in B_t by "winding-up" ρ. In a single iteration of the loop τ_t, the first part of the states of τ_t, $(s_1[0], \ldots, s_m[0]), \ldots, (s_1[\gamma| - 1] \ldots, s_m[\gamma| - 1]$, must be capable of holding all states of $\sigma[|\beta|, |\sigma|-1]$; the second part, $(l_1[0], \ldots, l_n[0]), \ldots, (l_1[|\gamma|-1], \ldots, l_n[|\gamma|-1])$, has to hold $lcm(|\tau||\gamma|)$ states. Hence, we set:

$$m = \lceil \frac{|\sigma|-|\beta|}{|\gamma|} \rceil \qquad\qquad n = \frac{lcm(|\tau|,|\gamma|)}{|\gamma|}$$

σ_t is straight-forward: only $s_1[i]$ is relevant and is set to $\sigma[i]$. $s_2[i], \ldots, s_m[i]$ can be set to "don't care", denoted "$-$", as can $l_1[i], \ldots, l_n[i]$.

$$\forall 0 \leq i < |\beta| \,.\, \sigma_t[i] = (\sigma[i], -, \ldots, -, (-, 0, \ldots, 0), \ldots, (-, 0, \ldots, 0), st)$$

We can now define τ_t as follows:

$$\begin{aligned}&\forall 0 \leq i < |\gamma| \,.\, \tau_t[i] = \\ &\quad (s_1[i], \ldots, s_i[i], (l_1[i], f_{1,1}[i], \ldots, f_{1,|F|}[i]), \ldots, (l_j[i], f_{j,1}[i], \ldots, f_{j,|F|}[i]), lo[i])\end{aligned}$$

with

$$\begin{aligned}
&\forall 0 \leq i < |\gamma| \,. \\
&\quad \forall 1 \leq p \leq m \,.\, s_p[i] = \begin{cases} \sigma[|\beta| + i + (p-1)|\gamma|] & \text{if} \quad p < m \vee i < (|\sigma| - |\beta|) mod |\gamma| \\ \tau[i - (|\sigma| - |\beta|) mod |\gamma|] & \text{otherwise} \end{cases} \\
&\wedge \forall 1 \leq q \leq n \,.\, l_q[i] = \tau[((i + |\gamma| - (|\sigma| - |\beta|) mod |\gamma|) mod |\gamma| + (q-1)|\gamma|) mod |\tau|] \\
&\wedge \forall 1 \leq q \leq n \,.\, \forall 1 \leq k \leq |F| \,. \\
&\quad f_{q,k}[i] \leftrightarrow (\exists \hat{q}, \hat{i} \,.\, ((1 \leq \hat{q} < q \wedge 0 \leq \hat{i} < |\gamma|) \vee (\hat{q} = q \wedge 0 \leq \hat{i} \leq i)) \wedge \\
&\qquad\qquad \tau[((\hat{i} + |\gamma| - (|\sigma| - |\beta|) mod |\gamma|) mod |\gamma| + (\hat{q}-1)|\gamma|) mod |\tau|] \in F_k) \\
&\wedge lo[i] = lb \leftrightarrow i \neq |\gamma| - 1 \wedge lo[i] = le \leftrightarrow i = |\gamma| - 1
\end{aligned}$$

□

Remarks The bounds in the above construction can be restricted significantly for some constructions to obtain Büchi automata. Theorem 24 shows that for GPVW [GPVW96] the excess length of the stem is linear in the operator depth and that the loop is as short as required.

5.5 Related Work

5.5.1 Virtual unrolling

Virtual unrolling has first been described by Benedetti and Cimatti [BC03]. The work most directly related to ours is the one by Latvala et al.[LBHJ05], which inspired the tight encoding. Compared to [BC03] the encoding in [LBHJ05] is simpler and generates smaller problems for the SAT solver. It has also been extended to be incremental and complete [HJL05]. Jonsson and Nilsson use vectors of states to construct a Büchi automaton (termed "history transducer"), which represents or approximates the transitive closure of the transition relation of an infinite state system in the context of regular model checking [JN00].

5.5.2 Tight automata

For comments on the original notion of tight automata see Sect. 2.7. Awedh and Somenzi remark the lack of tight Büchi automata in their approach to make bounded model checking complete [AS04]. Gastin et al. make the same observation in their work on finding shortest fair cycles with SPIN [GMZ04]. Awedh and Somenzi hint that using edge-labeled Büchi automata may produce shorter counterexamples [AS04]. We conjecture that there is a conversion between node- and edge-labeled Büchi automata that preserves tightness.

5.5.3 Translating PLTLB into automata

First approaches Wolper, Vardi, and Sistla first showed how to compile PLTLB directly into Büchi automata [WVS83, VW94]. Based on that, Vardi and Wolper proposed the automata-theoretic approach to model checking [VW86]. The tableau construction, which is a foundation for most of the following work, was presented by Lichtenstein and Pnueli as part of a practical model checking algorithm for linear temporal logic [LP85].

Focus on symbolic model checking In symbolic model checking, a compact symbolic representation of the automaton has mostly been preferred to a small number of states. Büchi automata for that purpose are usually symbolic implementations of the tableau construction in [LP85]. Burch et al. use a variant of the tableau to show how to reduce model checking of future time LTL to symbolic model checking of fair CTL [BCM+92]; implementation for SMV [McM93], proofs, and experimental evaluation of the approach are presented in [CGH97]. A self-contained presentation of symbolic model checking of PLTLB can be found in [KPR98]. Schneider and Schuele [Sch01, SS04] use the temporal hierarchy [MP90] to generate improved automata on infinite words and encodings for bounded model checking.

Focus on explicit state model checking The number of states in the product of the model and the Büchi automaton for the property determines to a large extent the amount of work an explicit state model checker has to do. As a consequence, there is much work that shows how to obtain smaller Büchi automata from a PLTLB formula. Gerth et al. [GPVW96] pioneered this line of research for future time LTL. A key to their approach is not to establish whether all subformulae hold at a certain state, but to track only a subset of subformulae needed to establish validity of the specification. As an example, it is not necessary to follow both sides of a disjunct (be it part of the original formula or due to the expansion of an $\mathbf{U}$-operator) at the same time or to care for the validity of $\mathbf{F}p$ in the initial state when the specification is $\mathbf{XF}p$. However, this makes it impossible for most constructions based on [GPVW96] to accept shortest counterexamples. Couvreur [Cou99] removes some of the formulae making up a state in [GPVW96]. Daniele et al. [DGV99] propose a general framework for algorithms based on [GPVW96] and present an improved algorithm. Somenzi and Bloem add a pre- and post-processing stage to the framework of Daniele et al. [SB00]. In the pre-processing stage they rewrite the PLTLB formula to obtain a smaller formula to start with. In the post-processing stage they use simulation to further reduce the size of the automaton. Etessami and Holzmann came up with a similar approach [EH00]. To improve performance of the translation itself, Gastin and Oddoux change the core of the algorithm by first translating into a very weak alternating automaton and only then into a Büchi automaton [GO01]. They also perform optimizations on the fly rather than in the post-processing stage to keep intermediate results small. They extend their approach to full PLTLB in [GO03]. Sebastiani and Tonetta focus on producing more deterministic rather than smaller Büchi automata to obtain a smaller product of model and Büchi automaton [ST03].

Translating safety properties Several authors consider translation of linear time safety properties into automata on finite words. For [KV01] see Sect. 2.7. Latvala implements an optimized translation for intentionally or accidentally safe formulae based on [KV01] that includes a check whether a formula is pathologically safe or not a safety formula at all [Lat03]. A similar, though less optimized approach is by Geilen [Gei01]. He produces automata to recognize both bad and good prefixes. The automated software engineering group at NASA Ames has developed several translations geared towards testing and monitoring executions (i.e., finite traces), e.g., [HR01a, GH01, HR02]. In [HR01a] and [GH01] different translations of future time LTL adapted to finite traces are presented. (Non-)occurrence of eventualities is only considered up to the end of a trace. Based on the belief that past time LTL is more appropriate for monitoring [HR01a], Havelund and Roşu present a translation from a version of past time LTL that has been extended with operators useful for monitoring execution traces [HR02]. Earlier approaches to monitoring include [HLR94, JPO95]. Both essentially limit support to formulae of the form $\mathbf{G}p$. Hence, the user must make sure that p represents an appropriate past formula implemented, e.g., with history variables.

Testing of translations Daniele et al. introduced a test method for the translation of linear temporal logic formulae into Büchi automata based on random formulae [DGV99]. Tauriainen and Heljanko present a comprehensive approach and implementation for LTL formula translation into Büchi automata [TH02].

5.6 Summary

We have extended the notion of a tight automaton by Kupferman and Vardi [KV01], which accepts shortest bad prefixes for safety properties, to Büchi automata. A necessary and sufficient criterion for a Büchi automaton to be tight is, that for each counterexample π in the language of the automaton there is a run ρ such that states in π with the same future are accepted from the same state in ρ (i.e., $\pi[i,\infty] = \pi[j,\infty] \Rightarrow \rho[i] = \rho[j]$). This was used to prove that a Büchi automaton constructed with the algorithm of Kesten et al. [KPR98] is tight for future time LTL but may produce counterexamples with excess length linear in the past operator depth of the property. The algorithm by Gerth et al. [GPVW96] may lead to counterexamples with excess length linear in the future operator depth. As the two most common approaches to construct Büchi automata do not lead to tight automata, we have, inspired by the work of Latvala et al. [LBHJ05], adapted virtual unrolling as introduced by Benedetti and Cimatti [BC03] for bounded model checking to Büchi automata. This lead to a translation from PLTLB to a tight automaton as well as to a more general construction to make an arbitrary Büchi automaton tight.

6
Variable Optimization

Optimization hinders evolution.
Alan Perlis

The overhead induced by the state-recording translation mostly stems from the additional instance of the state variables of K present in $K^{\mathbf{S}}$. Several variants of *variable optimization* try to alleviate that overhead by restricting loop detection to a subset of variables.

Intuition and Definition Assume some Kripke structure $K = (S, T, I, L, F)$ and let $V' \subseteq V$ be a subset of its state variables. Let $K^{\mathbf{S}|_{V'}}$ denote the variant of $K^{\mathbf{S}}$ that stores only a projection of the presumed loop start s_l onto the variables in V' and, correspondingly, compares only the projection of the current state s_k onto the variables in V' to the stored projection of s_l. As $K^{\mathbf{S}|_{V'}}$ has fewer state variables than $K^{\mathbf{S}}$, we can hope for improved performance. Definition 9 formally states the construction of $K^{\mathbf{S}|_{V'}}$. Note that variable optimization is source-to-source and can even be applied manually to $K^{\mathbf{S}}$.

Outline We start by proving the following fact: if no counterexample can be found using an arbitrary subset of state variables $V' \subseteq V$ for loop detection in $K^{\mathbf{S}|_{V'}}$, then clearly no counterexample is present in the original system K either. As removing arbitrary variables from loop detection may obviously lead to spurious counterexamples, we propose increasingly aggressive methods — i.e., smaller and smaller sets V' — that avoid spurious counterexamples. First, it's easy to see that constants and input variables can be removed without incurring spurious counterexamples. Then we show that cone of influence reduction carries over to variable optimization from the standard model checking algorithm as well. Finally, abstraction refinement is used to obtain a sound and complete algorithm for variable optimization with arbitrary sets V'.

6.1 The General Case

Result The following theorem states that if a property passes with a reduced set of variables V' in loop detection it will also pass when the full set of variables V is used:

Theorem 33 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure, let V be its set of state variables, and let $V' \subseteq V$ be a (potentially empty) subset of its state variables. With $K^{\mathbf{S}}$ as in*

Definition 9 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure and let $V' \subseteq V$ be a subset of its state variables. Let $\hat{s}_0 \in S|_{V'}$ be arbitrary but fixed. Then $K^{\mathbf{S}|_{V'}} = (S^{\mathbf{S}|_{V'}}, T^{\mathbf{S}|_{V'}}, I^{\mathbf{S}|_{V'}}, L^{\mathbf{S}|_{V'}}, F^{\mathbf{S}|_{V'}})$ is defined as:*

$$
\begin{array}{lll}
S^{\mathbf{S}|_{V'}} & = & S \times S|_{V'} \times \{st, lb, lc\} \times \mathbb{B} \\
I^{\mathbf{S}|_{V'}} & = & \{(s_0, \hat{s}_0, st, 0) \mid s_0 \in I\} \cup \\
 & & \{(s_0, s_0|_{V'}, lb, f) \mid s_0 \in I \wedge (f \rightarrow s_0 \in F_0)\} \\
T^{\mathbf{S}|_{V'}} & = & \{((s, \hat{s}, lo, f), (s', \hat{s}', lo', f')) \mid (s, s') \in T \wedge \\
 & & \quad ((lo = st \wedge lo' = st \wedge \neg f \wedge \neg f' \wedge \hat{s} = \hat{s}' = \hat{s}_0) \vee \qquad (1) \\
 & & \quad\; (lo = st \wedge lo' = lb \wedge \neg f \wedge (f' \rightarrow s' \in F_0) \wedge \hat{s} = \hat{s}_0 \wedge (s'|_{V'}) = \hat{s}') \vee \qquad (2) \\
 & & \quad\; (lo = lb \wedge lo' = lb \wedge (f \rightarrow f') \wedge (f' \rightarrow f \vee s' \in F_0) \wedge \hat{s} = \hat{s}') \vee \qquad (3) \\
 & & \quad\; (lo = lb \wedge lo' = lc \wedge f \wedge f' \wedge \hat{s} = (s'|_{V'}) = \hat{s}') \vee \qquad (4) \\
 & & \quad\; (lo = lc \wedge lo' = lc \wedge f \wedge f' \wedge \hat{s} = \hat{s}'))\} \qquad (5) \\
L^{\mathbf{S}|_{V'}}((s, \hat{s}, lo, f)) & = & L(s) \\
F^{\mathbf{S}|_{V'}} & = & \emptyset
\end{array}
$$

Def. 1 and $K^{\mathbf{S}|_{V'}}$ as in Def. 9 we have

$$R(K^{\mathbf{S}|_{V'}}) \cap \{s^{\mathbf{S}|_{V'}} \in S^{\mathbf{S}|_{V'}} \mid lc(s^{\mathbf{S}|_{V'}})\} = \emptyset \Rightarrow R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\} = \emptyset$$

Proof: We prove the reverse direction. Assume $R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\} \neq \emptyset$. Hence, there is an initialized path $\pi^{\mathbf{S}} = (s_0, \hat{s}_0, lo_0, f_0) \ldots (s_k, s_l, lc, 1)$ to some $(s_k, s_l, lc, 1) = s^{\mathbf{S}} \in R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\}$. It's now easy to see that $\pi^{\mathbf{S}}$ with its second component projected onto V' is an initialized path in $K^{\mathbf{S}|_{V'}}$ to $(s_k, s_l|_{V'}, lc, 1) = s^{\mathbf{S}|_{V'}} \in R(K^{\mathbf{S}|_{V'}}) \cap \{s^{\mathbf{S}|_{V'}} \in S^{\mathbf{S}|_{V'}} \mid lc(s^{\mathbf{S}|_{V'}})\}$. □

Variable optimization as existential abstraction Variable optimization is an instance of existential abstraction [CGL94] or, more specifically, variable projection (e.g, [BGG02]). Hence, for the proof of Thm. 33 we could have simply appealed to Corollary 5.7 in [CGL94]. We preferred to give the direct proof because of its simplicity.

6.2 Removing Constants

Constants after initialization A variable v is *constant after initialization* if its value doesn't change after the initial state: $\forall (s, s') \in T \,.\, v(s) = v(s')$.

Result The following theorem states the obvious fact that such constants need neither be stored nor compared in the state-recording translation.

Theorem 34 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure, let V be its set of state variables, and let $V_c \subseteq V$ be its set of variables constant after initialization. With $K^{\mathbf{S}}$ as in Def. 1 and $K^{\mathbf{S}|_{V \setminus V_c}}$ as in Def. 9 we have*

$$R(K^{\mathbf{S}|_{V \setminus V_c}}) \cap \{s^{\mathbf{S}|_{V \setminus V_c}} \in S^{\mathbf{S}|_{V \setminus V_c}} \mid lc(s^{\mathbf{S}|_{V \setminus V_c}})\} \neq \emptyset \Leftrightarrow R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\} \neq \emptyset$$

In addition, if $s^{\mathbf{S}} \in S^{\mathbf{S}}$ with $lc(s^{\mathbf{S}})$ true is reachable via $\pi^{\mathbf{S}}$ in $K^{\mathbf{S}}$ then some $s^{\mathbf{S}|_{V \setminus V_c}} \in S^{\mathbf{S}|_{V \setminus V_c}}$ with $lc(s^{\mathbf{S}|_{V \setminus V_c}})$ true is reachable in $K^{\mathbf{S}|_{V \setminus V_c}}$ via $\pi^{\mathbf{S}|_{V \setminus V_c}}$ such that $\pi^{\mathbf{S}}$ and $\pi^{\mathbf{S}|_{V \setminus V_c}}$ agree on all state variables present in both, $K^{\mathbf{S}}$ and $K^{\mathbf{S}|_{V \setminus V_c}}$.

Proof: Trivial. □

Identifying constants after initialization In our experiments we syntactically identify a variable as a constant after initialization

1. if it is unconditionally assigned its current state value as its next state value in an `ASSIGN` statement, or
2. if it is unconditionally assigned its current state value as its next state value in a `TRANS` statement, or
3. if it is constrained by an `INVAR` statement to a constant value.

We do not look for constants after initialization in the Büchi automaton representing the property.

6.3 Removing Input Variables

Kroening and Strichman proved in the context of bounded model checking [BCCZ99] that input variables can be ignored when computing the recurrence diameter for simple liveness properties of the form $\mathbf{F}p$ [KS03]. Eén and Sörensson [ES03] use the same idea in temporal induction for safety properties in incremental bounded model checking [Sht01, WKS01]. We now extend this idea to the state-recording translation.

Transition input variables A variable v is a *transition input variable* if its value in the next state is neither constrained by its value in the current state nor by the values of other variables in the current and next states:

$$\forall (s, s') \in T \;.\; \forall x \text{ in the range of } v \;.\; \exists (s, s'') \in T \;.\; s'|_{V \setminus v} = s''|_{V \setminus v} \wedge v(s'') = x$$

Note that, contrary to what would be expected for a "proper" input variable, we don't make any assumptions on the potential values of v in an initial state.

Result Intuitively, we can ignore transition input variables in the state-recording translation because we can set them to an arbitrary value (and, hence, to the same value as in s_l) when closing the loop in s_k:

Theorem 35 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure, let V be its set of state variables, and let $V_i \subseteq V$ be its set of transition input variables. With $K^{\mathbf{S}}$ as in Def. 1 and $K^{\mathbf{S}|_{V \setminus V_i}}$ as in Def. 9 we have*

$$R(K^{\mathbf{S}|_{V \setminus V_i}}) \cap \{s^{\mathbf{S}|_{V \setminus V_i}} \in S^{\mathbf{S}|_{V \setminus V_i}} \mid lc(s^{\mathbf{S}|_{V \setminus V_i}})\} \neq \emptyset \Leftrightarrow R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\} \neq \emptyset$$

In addition, if $s^{\mathbf{S}} \in S^{\mathbf{S}}$ *with* $lc(s^{\mathbf{S}})$ *true is reachable via* $\pi^{\mathbf{S}}$ *in* $K^{\mathbf{S}}$ *then some* $s^{\mathbf{S}|_{V \setminus V_i}} \in S^{\mathbf{S}|_{V \setminus V_i}}$ *with* $lc(s^{\mathbf{S}|_{V \setminus V_i}})$ *true is reachable in* $K^{\mathbf{S}|_{V \setminus V_i}}$ *via* $\pi^{\mathbf{S}|_{V \setminus V_i}}$ *such that* $\pi^{\mathbf{S}}$ *and* $\pi^{\mathbf{S}|_{V \setminus V_i}}$ *agree on all state variables present in both,* $K^{\mathbf{S}}$ *and* $K^{\mathbf{S}|_{V \setminus V_i}}$.

Proof: The "$\Leftarrow$"-direction of the first claim and the second claim follow from the proof of Thm. 33. For "$\Rightarrow$" note that in Def. 9 (as in the unoptimized case, see Def. 1) fair states need to be seen strictly before the loop can be closed. Hence, it is sufficient to prove the following implication: if $\widetilde{\pi} = s_0 \ldots s_l \ldots \widetilde{s_k}$ is an initialized finite path in K with $k > l > 0$ and $v(\widetilde{s_k}) = v(s_l)$ for all variables $v \in V \setminus V_i$, then $\widetilde{\pi}$ with its last state replaced by s_l is an initialized finite path in K with $k > l > 0$ and $s_k = s_l$. By assumption, $(s_{k-1}, \widetilde{s_k}) \in T$. Construct a sequence of states $\widetilde{s_k} = t_0, t_1, \ldots, t_{|V_i|} = s_l$ such that all t_j, t_{j+1} differ at most by the value of one variable in V_i. By definition, for each t_j, t_{j+1}, $(s_{k-1}, t_j) \in T$ iff $(s_{k-1}, t_{j+1}) \in T$. Hence, $(s_{k-1}, s_l) \in T$. $\square$

Transition input variables in the property If the Kripke structure being transformed is the product of a model M and a Büchi automaton B generated from a PLTLB formula, the set of transition input variables has to be determined with respect to the product $M \times B$. Hence, input variables of M that appear in the PLTLB formula to be verified may need to be included in the loop detection.

Identifying transition input variables In our experiments we use a syntactic approach to conservatively identify transition input variables. A transition input variable may not appear in either of the following contexts:

1. an `INVAR` statement,
2. a `DEFINE` statement,
3. in the scope of a `next` operator in an `ASSIGN` statement, or
4. in the scope of a `next` operator in a `TRANS` statement.

We make an exception to these rules for the `_process_selector_` system variable and sometimes use knowledge of the model to identify more input variables. We do not look for transition input variables in the Büchi automaton representing the property.

6.4 Cone of Influence Reduction for Loop Detection

Baumgartner et al. [BKA02] observed that the diameter of a system need only be computed for the variables in the cone of influence (e.g., [CGP99]) of the property. This idea carries over to the state-recording translation.

The fair cone of influence The *fair cone of influence* of a Kripke structure $K = (S, T, I, L, F)$ w.r.t. some PLTLB formula ϕ is the set of all variables that influence the behavior of K w.r.t. fairness and variables occurring in ϕ. Formally, $V_{coi} \subseteq V$ is the smallest set containing all variables occurring in ϕ and fulfilling

$$\begin{aligned}\forall s_1, s_2 \in S \,.\, (s_1|_{V_{coi}} = s_2|_{V_{coi}} \Rightarrow \\ &(\forall s_1' \in S \,.\, ((s_1, s_1') \in T \Rightarrow \exists s_2' \in S \,.\, (s_1'|_{V_{coi}} = s_2'|_{V_{coi}} \wedge (s_2, s_2') \in T))) \wedge \\ &(\forall 0 \leq m \leq f \,.\, (s_1 \in F_m \Leftrightarrow s_2 \in F_m)))\end{aligned}$$

Result Theorem 36 shows that only variables in the fair cone of influence need to be considered in loop detection. Additional intuition for the correctness of the theorem can be gleaned from the possibility to apply standard cone of influence reduction before the state-recording translation.

Theorem 36 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure, let V be its set of state variables, and let $V_{coi} \subseteq V$ be its fair cone of influence. With $K^{\mathbf{S}}$ as in Def. 1 and $K^{\mathbf{S}|_{V_{coi}}}$ as in Def. 9 we have*

$$R(K^{\mathbf{S}|_{V_{coi}}}) \cap \{s^{\mathbf{S}|_{V_{coi}}} \in S^{\mathbf{S}|_{V_{coi}}} \mid lc(s^{\mathbf{S}|_{V_{coi}}})\} \neq \emptyset \Leftrightarrow R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\} \neq \emptyset$$

Proof: Again, the implication from right to left follows from Thm. 33. For "$\Rightarrow$" assume that some $s^{\mathbf{S}|_{V_{coi}}} \in S^{\mathbf{S}|_{V_{coi}}}$ with $lc(s^{\mathbf{S}|_{V_{coi}}})$ true is reachable in $K^{\mathbf{S}|_{V_{coi}}}$. Let $\pi^{\mathbf{S}|_{V_{coi}}} = (s_0, \hat{s}_0, st, 0) \ldots (s_l, s_l|_{V_{coi}}, lb, 0) \ldots (s_m, s_l|_{V_{coi}}, lb, 1) \ldots (s_k, s_l|_{V_{coi}}, lc, 1)$ with $k > m \geq l \geq 0$, $s_m \in F_0$, and $s_k|_{V_{coi}} = s_l|_{V_{coi}}$ be a finite initialized path leading to $s^{\mathbf{S}|_{V_{coi}}} = (s_k, s_l|_{V_{coi}}, lc, 1)$.* We inductively show that $s_0 \ldots s_l \ldots s_m \ldots s_k$ can be extended to an infinite initialized path π in K with $\pi|_{V_{coi}} = (s_0 \ldots s_{l-1}(s_l \ldots s_m \ldots s_{k-1})^\omega)|_{V_{coi}}$. With $s_m \in F_0$ and by the definition of V_{coi} we have that $\forall n \geq 0 \,.\, \pi[m + n(k-l)] \in F_0$ and, hence, any such π is fair. The claim then follows from Thm. 5. The base case is given by $\pi[0, k] = s_0 \ldots s_l \ldots s_m \ldots s_k$ being a finite initialized path in K with $\pi[k]|_{V_{coi}} = \pi[l]|_{V_{coi}}$ according to Def. 9. For the inductive case let $\pi[0, i]$ with $i \geq k$ be an extension of $\pi[0, k]$ in K such that $\pi[0, i]|_{V_{coi}} = (s_0 \ldots s_{l-1}(s_l \ldots s_m \ldots s_k)^\omega)[0, i]|_{V_{coi}}$. By inductive assumption $\pi[i]|_{V_{coi}} = \pi[i - (k - l)]|_{V_{coi}}$. Hence, with the definition of V_{coi}, there is s_{i+1} such that $s_{i+1}|_{V_{coi}} = \pi[i - (k - l) + 1]|_{V_{coi}}$ and $(\pi[i], s_{i+1}) \in T$. Clearly, $(\pi[0, i] \circ s_{i+1})|_{V_{coi}} = (s_0 \ldots s_{l-1}(s_l \ldots s_m \ldots s_k)^\omega)[0, i+1]|_{V_{coi}}$. □

Combination with removal of constants and transition input variables It's not hard to see that Thms. 34, 35, and 36 can be combined:

Theorem 37 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure with a total transition relation T. Let V be its set of state variables, $V_c \subseteq V$ its variables constant after initialization, $V_i \subseteq V$ its set of transition input variables, and V_{coi} its fair cone of influence. With $V' = V_{coi} \setminus (V_c \cup V_i)$, $K^{\mathbf{S}}$ as in Def. 1, and $K^{\mathbf{S}|_{V'}}$ as in Def. 9 we have*

$$R(K^{\mathbf{S}|_{V'}}) \cap \{s^{\mathbf{S}|_{V'}} \in S^{\mathbf{S}|_{V'}} \mid lc(s^{\mathbf{S}|_{V'}})\} \neq \emptyset \Leftrightarrow R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\} \neq \emptyset$$

Proof: The "$\Leftarrow$"-direction follows from Thm. 33. We only sketch the reverse direction. Start by assuming an initialized path $\pi^{\mathbf{S}|_{V'}} = (s_0, \hat{s}_0, st, 0) \ldots (s_l, s_l|_{V'}, lb, 0) \ldots (s_m, s_l|_{V'}, lb, 1) \ldots (s_k, s_l|_{V'}, lc, 1)$ with $k > m \geq l \geq 0$, $s_m \in F_0$, and $s_k|_{V'} = s_l|_{V'}$ leading to some violating state $s^{\mathbf{S}|_{V'}} = (s_k, s_l|_{V'}, lc, 1)$. Transform s_k into $\tilde{s}_k$ as in the proof of Thm. 35 such that $s_l|_{V_{coi}} = \tilde{s}_k|_{V_{coi}}$. Continue from there with $\pi[0, k] = s_0 \ldots \tilde{s}_k$ as in the proof of Thm. 36. □

*Note that we are a bit sloppy here: in principle we'd have to establish that $s^{\mathbf{S}|_{V_{coi}}}$ and $\pi^{\mathbf{S}|_{V_{coi}}}$ must be of that shape as in the proof of Thm. 5. To avoid unnecessary detail we refer the reader to that proof instead.

Finding the fair cone of influence For the experiments we again use a syntactic scheme to identify the cone of influence. A variable `v` depends on another variable `w` if one of the following conditions is met:

1. `v` and `w` appear in the same `INIT` statement,
2. `v` and `w` appear in the same `INVAR` statement,
3. `v` and `w` appear in the same `TRANS` statement and `v` is in the scope of a `next` operator, or
4. `v` and `w` appear in the same `ASSIGN` statement and `v` is in the scope of an `init` or `next` operator.

The fair cone of influence is defined as the smallest set V_{coi} that contains (1) the variables appearing in the property and in fairness constraints and (2) all variables on which those in V_{coi} depend. All variables in the Büchi automaton representing the property are considered part of the fair cone of influence. In addition, the transition relation of K must be required to be total.

Influence on counterexamples The previous variable optimizations require hardly any extra effort to reconstruct a counterexample in K from one in $K^{\mathbf{S}|_{V'}}$. In particular, shortest counterexamples can be found (see Sect. 3.4). As when standard cone of influence reduction is applied to the system K, a counterexample found in $K^{\mathbf{S}|_{V'}}$ may need some reconstruction effort to obtain a counterexample in the full system K. Specifically, its stem and/or loop may need to be extended. Therefore, shortest counterexamples cannot be guaranteed. As an aside, note that classical cone of influence reduction has to be applied to K rather than to $K^{\mathbf{S}}$; otherwise, all variables being stored and compared will end up in the cone of influence.

6.5 Abstraction Refinement for Loop Detection

Motivation Abstraction refinement has already proven to be an effective means for combatting state explosion (e.g., [CGJ+03]), especially so when the property is true [BGG02]. Experimental results (see Chap. 7) show that the overhead of the state-recording translation is considerably higher when a property passes than when it fails. The previous instances of variable optimization made sure that both a passing and a failing result from the optimized system carries over to the unoptimized system. To investigate whether removing more variables from loop detection at the price of admitting spurious counterexamples might help, we implemented a simple abstraction refinement scheme for the set of variables used in loop detection.

Algorithm Figure 6.1 shows the algorithm, which follows the general scheme outlined in Fig. 2.2. $check\_reachable(K, S')$ determines whether a state in S' is reachable in K. If yes, a 0 result and a path leading to such state are returned. Otherwise, the result is 1 and an empty path. The first line assigns a set of state variables to V_{max} such that no spurious counterexamples can occur; we chose $V_{max} = V_{coi} \setminus (V_c \cup V_i)$ based on Thm. 37. In line 2, V' is initialized with an arbitrary subset of V_{max} — we start with $V' = \emptyset$. The algorithm then applies Thm. 33 (line 4). If no counterexample is found in line 4, we are done (lines 5, 6). Otherwise, if $V' = V_{max}$,

Require: a fair Kripke structure $K = (S, T, I, L, F)$ with state variables V
Ensure: return 1 iff $Lang(K) = \emptyset$
1: let V_{max} such that $V' = V_{max}$ prevents spurious counterexamples
2: let $V' \subseteq V_{max}$
3: **loop**
4: let $(result, \pi^{\mathbf{S}|_{V'}}) := check\_reachable(K^{\mathbf{S}|_{V'}}, \{s^{\mathbf{S}|_{V'}} \in S^{\mathbf{S}|_{V'}} \mid lc(s^{\mathbf{S}|_{V'}})\})$
5: **if** $result = 1$ **then**
6: return 1
7: **else if** $V' = V_{max}$ **then**
8: return 0
9: **else if** $s(\pi^{\mathbf{S}|_{V'}}[l]) = s(\pi^{\mathbf{S}|_{V'}}[k])$ **then**
10: return 0
11: **else**
12: let $V' := V' \cup V''$ for some $\emptyset \subset V'' \subseteq V_{max} \setminus V'$
13: **end if**
14: **end loop**

Figure 6.1: Abstraction refinement for loop detection

the counterexample must be real (lines 7, 8). If $V' \subset V_{max}$ a quick check is made in line 9 to determine whether the counterexample is real despite $V' \neq V_{max}$ and 0 is returned if the check is successful (the algorithm in Fig. 6.1 would be sound and complete without this step). If not, the counterexample must be spurious. Hence, we add some variables from $V_{max} \setminus V'$ to V' according to the following scheme (lines 11, 12): If $V' = \emptyset$, add all variables appearing in the property and in fairness constraints; otherwise, add all variables on which the variables already contained in V' depend. Now we repeat the loop with a strictly larger set V'.

The algorithm makes only a very limited attempt to reconstruct a counterexample in the unoptimized system. It also follows a static refinement scheme based on the dependency relation of the state variables rather than analyzing a spurious counterexample. Still, it turns out to be quite helpful to improve performance in problematic cases.

Result Theorem 38 states that the algorithm is sound and complete.

Theorem 38 *Let $K = (S, T, I, L, F = \{F_0\})$ be a fair Kripke structure. The algorithm in Fig. 6.1 terminates and returns 1 iff*

$$R(K^{\mathbf{S}}) \cap \{s^{\mathbf{S}} \in S^{\mathbf{S}} \mid lc(s^{\mathbf{S}})\} \neq \emptyset$$

Proof: Correctness follows from Thm. 33 (line 6), the assumption in line 1 (line 8), and Def. 9 (line 10). Line 12 and finiteness of V_{max} give termination. □

6.6 Utility of ...

... removing constants and input variables Removing constants and input variables in the state-recording translation as shown in Thms. 34, 35 is independent of reductions such as standard cone of influence reduction [CGP99] or more general instances of existential abstraction

```
MODULE model
VAR
  a: boolean;
  b: boolean;
INIT
  a
TRANS
  a -> next(b)
```

```
MODULE buechi(b)
VAR
  s:{s0};
INVAR
  (s=s0) -> !b
FAIRNESS
  s=s0
```

```
MODULE main
VAR
  mo: model;
  ba: buechi(mo.b);
SPEC
  !EG 1
```

Figure 6.2: This example for SMV shows that the state-recording translation can prove a property to be true with an empty set of variables in loop detection.

[CGL94] typically performed on the system as a whole. Hence, given noticeable performance benefits (see Chap. 7) and absence of influence on the length of a potential counterexample, these should always be applied.

... removing the fair cone of influence The situation is different when removing variables not in the cone of influence of the property (Thm. 36): one can equally well perform standard cone of influence reduction on the system before applying the state-recording translation and expect greater impact on performance.[†] We have included it here to investigate the performance impact of removing more variables than just constants and input variables and, more importantly, as an easy-to-get upper bound for the abstraction refinement algorithm in Fig. 6.1.

... using abstraction refinement Similar reservations could be brought forward w.r.t. the algorithm in Fig. 6.1: why not apply abstraction refinement to the original system and the state-recording translation to the abstracted versions of the system. However, our scheme is not only an intuitive example to study the performance of more aggressive variants of variable optimization, but is also useful in its own right. Consider the example in Fig. 6.2 in the language of SMV [McM93, CCO+]. We wish to verify that $\mathbf{F}b$ holds in the module `model`. Hence, module `buechi` encodes a Büchi automaton accepting witnesses to $\mathbf{G}\neg b$. The system can be verified after applying the state-recording translation with an empty set of variables used for loop detection, while the system abstracted to the variables `mo.b` and `ba.s` has a spurious counterexample. In general, variable optimization with $V' = \emptyset$ may succeed to prove that a property ϕ holds if each initialized path either does not fulfill all fairness constraints or finishes being a finite informative witness [KV01] to ϕ before all fairness constraints have been fulfilled.

6.7 Related Work

6.7.1 Completeness in bounded model checking

Relation to simple path constraint Part of the inspiration for variable optimization and probably the most closely related idea comes from completeness of bounded model checking [BCCZ99]: a standard method to achieve completeness for BMC is checking in regular

[†]Due to time constraints we have not performed an experimental evaluation of this claim.

intervals whether the current bound k exceeds the length of any potential shortest counterexample in the model [BCCZ99, SSS00, BKA02, ES03, KS03, CKOS05, AS04, HJL05, AS06]. The corresponding constraint, often termed *simple path* after [AS04], requires that the current bound k allows for a loop-free path of length $k+1$ in the model. Here, "loop-free" is to be understood in a wide sense. The fixed subset of variables V' used to check whether two states of a path should be considered equal and, therefore, close a loop includes not only variables of the model but also those representing property and fairness constraints. The proof of correctness for a particular simple path constraint usually involves the following sequence of steps: (1) assume some "shortest" counterexample π not fulfilling the simple path constraint, (2) identify two states $\pi[i]$, $\pi[j]$ that must agree on a fixed subset V' of the state variables, and (3) derive a shorter counterexample $\tilde{\pi}$ by continuing after $\pi[i]$ with $\pi[j+1,\infty]$ (which might have to be suitably modified). This proves the claim by contradiction. Step (3) is very similar to variable optimization: given that s_l and s_k agree on a subset of state variables V' it must be established that continuing with $(s_l \ldots s_{k-1})^\omega$ after $s_0 \ldots s_{k-1}$ leads to a counterexample. A reduced set of state variables is used in the simple constraint by, e.g., [BKA02, KS03, ES03].

Specific works Baumgartner et al. ignore input variables and variables outside the cone of influence when computing the diameter of netlists [BKA02].

Kroening and Strichman proved that input variables can be ignored when computing the recurrence diameter for simple liveness properties of the form $\mathbf{F}p$ [KS03]. They also ignore an input variable v if v appears in the property. That does not extend to arbitrary properties: consider a model with just two input variables, a Boolean trigger t and an integer i. The recurrence diameter is 0, but the shortest witness for $\mathbf{G}(t \Rightarrow (\mathbf{F}(i=0) \wedge \mathbf{F}(i=1) \wedge \ldots \wedge \mathbf{F}(i=n))) \wedge \mathbf{GF}t$ has n states. They also show that the bounded cone of influence [BCRZ99] can be used to obtain an even smaller recurrence diameter. Implementing this construction for the state-recording translation requires introducing a counter and using a more complicated formulation of loop closure. Furthermore, the construction only affects the comparison of the current and the saved states; it is still necessary to save all variables in the cone of influence of the presumed loop start. Hence, we doubt that a BDD-based implementation of the state-recording translation would benefit much.

Eén and Sörensson remove input and output variables from their simple path constraint when they perform temporal induction for safety properties in incremental BMC [ES03].

Influence of removing input variables on termination depth Removing input variables may exponentially decrease the depth at which a property is proved in [KS03, ES03]. We conjecture that only a linear reduction can be achieved with our method. Note, though, that the main purpose of our reduction is to achieve smaller BDDs rather than a decrease in termination depth.

Output variables as a special case If output variables that do not appear in the property are removed from loop detection in the state-recording translation, shortest counterexamples cannot be guaranteed anymore. Note that such output variables are not part of the cone of influence; hence, they are removed with other variables not in the cone of influence when Thm. 36 is applied. The example in Fig. 6.3 shows how ignoring output variables may shorten a counterexample. The shortest counterexample in the unoptimized system is $(0,0)(1,0)(1,1)^\omega$ (states are denoted as (x,y)). Leaving y out of loop detection gives $(0)(1)^\omega$. On the other hand, such coun-

```
MODULE model
VAR
  x: boolean;
  y: boolean;
ASSIGN
  init(x) := 0;
  next(x) := 1;
  init(y) := 0;
  next(y) := x;
```

```
MODULE buechi(x)
VAR
  s:{s0,s1};
INVAR
  s=s1 -> x=1
TRANS
  s=s1 -> next(s)=s1
FAIRNESS
  s=s1
```

```
MODULE main
VAR
  mo: model;
  ba: buechi(mo.x);
SPEC
  !EG 1
```

Figure 6.3: This example for SMV shows that removing output variables from loop detection in the state-recording translation can lead to shorter counterexamples in the optimized system than in the full system.

terexample can always be extended to a full counterexample by letting the loop start and end with one state delay, as then all variables on which the output variables depend have stabilized.

Removing output variables from loop detection that do appear in the property may lead to incorrect results in the state-recording translation; Eén and Sörensson have a corresponding restriction.

6.7.2 Identifying input variables and variable dependencies

Papers with strong roots in hardware verification [BKA02, KS03] typically assume that input variables are a separate syntactic entity and that the next state value of a state variable is a function of the current state and the input. This makes identification of input variables, dependent variables and the cone of influence pretty straightforward. The original SMV [McM93] does not allow a variable to be declared as input variable. While NuSMV [CCG+02] added that feature, many benchmarks were written either before its introduction or refrain from using it for compatibility reasons. Hence we needed to devise criteria to identify input variables and variable dependencies based on a relational representation of the system. Eén and Sörensson took a similar approach [ES03]. They assume that the system is given as propositional formulas that describe the set of initial states and the transition relation. Input variables may occur neither in the initial state formula nor in the scope of a next operator in the transition relation.

6.7.3 Abstraction and refinement

Abstraction and automated abstraction refinement have been widely investigated. The following discussion can therefore be only selective; for more related work see, e.g., [CGJ+03]. As has been mentioned, variable optimization is an instance of existential abstraction [CGL94], more specifically, variable projection (e.g., [BGG02]). This is similar to removing entire processes or subsystems as in [BSV93, Kur94, LNA99] or (less useful in our case) cutting connections between some subsystems [LPJ+96] as well as using overlapping variable projection [GD98]. We focus mostly on the approaches just mentioned below. In particular we do not cover predicate abstraction [GS97] and the (highly successful) related automated refinement approaches [BR02, HJMS02, CCG+04].

Over- and under-approximation Existential abstraction over-approximates the reachable set of states. Lee et al. use under-approximations to obtain definite results for failing properties [LPJ+96]. Lind-Nielsen and Andersen combine over- and under-approximations to cover all of CTL [LNA99]. Note that in our case under-approximation by universal abstraction of variables with a range of size larger than 1 makes no sense: as we compare for equality no state such that lc is true would be reachable.

Automated abstraction refinement Balarin and Sangiovanni-Vincentelli [BSV93] and Kurshan [Kur94] were among the first to present automated abstraction refinement schemes. They focus on removing entire components or components and selected connections between components. Clarke et al. [CGJ+03] extended the principle to the more general framework of [CGL94].

Reconstruction We only perform a simplistic reconstruction to see whether the counterexample in the abstract corresponds to a real counterexample: we have a finite path in the original system and only check whether it happens to close a loop between two particular states s_l and s_k. We could also try whether any state is reachable from $s_{k-1}^{\mathbf{S}}$ such that $lc(s_k^{\mathbf{S}})$ is true and $s(s_k^{\mathbf{S}}) = s(s_l^{\mathbf{S}})$. Other approaches abstract the system; hence, reconstruction requires more effort. A typical approach performs forward reachability from the initial states or backward reachability from the bad states in the unabstracted system and intersects the states reached in each step with the corresponding step in the abstract counterexample (e.g., [Kur94, CGL94]). When a bad state (for forward reachability) or an initial state (for backward reachability) is hit, the unabstracted counterexample must be real. Otherwise, an empty intersection between the states reached in the system and the abstract counterexample will occur at some point. As reconstruction is also subject to state explosion, some approaches proceed iteratively here as well [BSV93, BGG02]. When both, over- and under-approximations are used, reconstruction may not be required at all [LPJ+96, LNA99].

Refinement We use a static refinement strategy that follows the dependencies between state variables without looking at the spurious counterexample. The sets of variables we add in an iteration correspond to the sets of variables with a fixed distance from the variables occurring in the specification or fairness constraints in the graph induced by the dependency relation. The same strategy is used by Lind-Nielsen and Andersen [LNA99]. The sets of variables used in an iteration correspond to the bounded cone of influence as employed for optimizations in bounded model checking [BCRZ99, KS03]. Balarin and Sangiovanni-Vincentelli add processes also in the order of their dependency relation but only until a counterexample has been removed. We could also only add one or more of the variables that are different in $s(s_k^{\mathbf{S}}) = s(s_l^{\mathbf{S}})$. Lee et al. follow a greedy strategy: they tentatively add each subsystem and finally choose the one which gives the largest reduction in terms of undesired states [LPJ+96]. Clarke et al. extend automated refinement to more general abstractions by partitioning the state in the abstract counterexample that can not be continued in the unabstracted system [CGJ+03]. Many authors also have the cone of influence as an upper bound of refinement [BSV93, Kur94, LNA99, BGG02]. Govindaraju and Dill [GD98] and Lind-Nielsen and Andersen [LNA99] reuse results from previous iterations. We could restart subsequent iterations from the set of states in which no loop start has been guessed yet.

Explicit state model checking SPIN offers a command line switch to store only hash values rather than entire states on the depth-first search stack to reduce memory consumption, though at the price of decreased performance [Hol03]. This is normally used as an add-on to the remotely related bitstate hashing, which has been introduced by Holzmann to reduce the amount of memory required to store the set of reached states during state space traversal [Hol88, Hol03]. While bitstate hashing may miss some states during state space traversal, the coverage achieved when using it is typically higher than what could be achieved without due to limited memory resources [Hol98].

Abstraction (refinement) in loop detection We are not aware of any work directly applying abstraction or abstraction refinement only to loop detection other than the SPIN command line switch just mentioned. As an example bounded model checking could apply a similar optimization for the loop closing condition.

6.8 Summary

Variable optimization targets the most important source of overhead in the state-recording translation by selectively removing variables from loop detection. Constants and input variables can be both ignored while guaranteeing that shortest counterexamples are found. Experimental results indicate no adverse effects, so these optimizations should always be enabled when using the state-recording translation. Removing all variables outside the cone of influence from loop detection sacrifices shortest counterexamples, but further improves performance. It is mainly useful as an upper bound for the most aggressive variant of variable optimizations presented here, which applies an abstraction refinement scheme to variable optimization: starting with few or even no variables, the set of variables used for loop detection is increased until either the property is proven to hold or a real counterexample is found. For properties that turn out to be true the gain in performance can be more than 2 orders of magnitude.

7 Experiments

Facts are the enemy of truth.
Miguel de Cervantes Saavedra, Don Quixote

In this chapter we evaluate the practical benefits of the state-recording translation and of tight Büchi automata in BDD-based symbolic model checking. A toy example demonstrates a potential exponential speed-up of forward reachability checking with the state-recording translation in comparison to a classical model checking algorithm. More realistic figures are then obtained on real-world examples for both time and memory usage. The length of counterexamples reported by classical model checking and by the state-recording translation is compared. SAT-based bounded model checking as an alternative method to find shortest counterexamples is evaluated against BDD-based reachability checking with the state-recording translation. Impact on resource usage of using a tight encoding and of performing variable optimization is evaluated. Finally, using tight automata with a classical symbolic model checking algorithm is examined.

7.1 A Forward Jumping Counter

The example Our translation may lead to a model that can be verified exponentially faster. Consider the n-bit counter shown in Fig. 7.1. It can jump forward from state i to an arbitrary state $j > i$. Only in the last state p is true. For the correct version $\mathbf{F}p$ holds, self-loops are added to generate an erroneous version. A standard algorithm for symbolic model checking [BCM+92] needs $\mathbf{O}(2^n)$ backward iterations to verify the correct counter. If the state-recording translation is applied, a constant number of forward iterations suffices as $r, d \leq 2$. Note that the experiment in this section was performed using the translation for simple liveness properties as shown in Fig. 3.3 (c). However, the results are also valid if Def. 1 is employed.

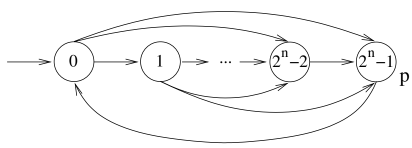


Figure 7.1: Forward jumping counter

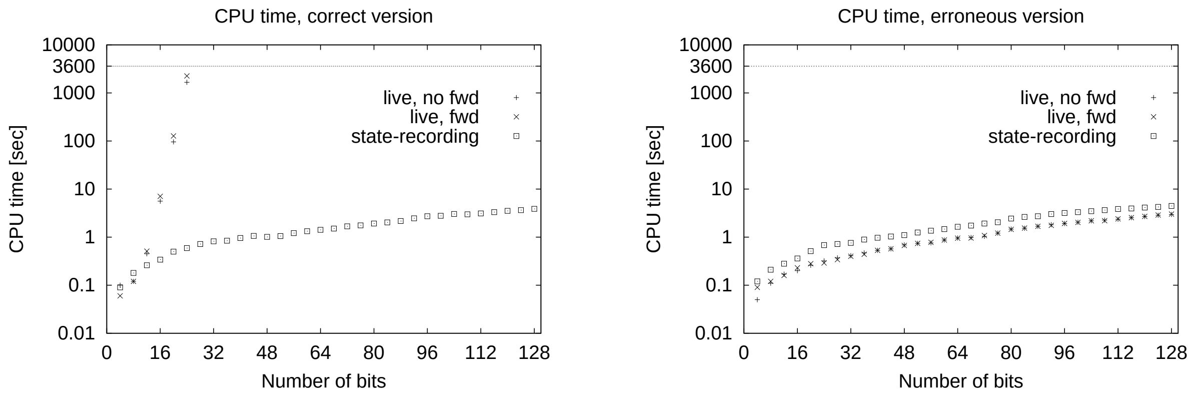


Figure 7.2: Forward jumping counter — state-recording translation versus standard approach versus forward model checking

Platform and resource bounds We used the model checker of the VIS system (v. 1.4) [VIS96] to verify the forward jumping counter. Apart from backward (standard) symbolic model checking [BCM+92] *VIS* also provides an implementation of the symbolic forward model checking algorithm by Iwashita et al. [INH96]. The experiments were performed on an Intel PC running at 800 MHz with 1.5 GB RAM, a wall clock limit was set at one hour.

Results The results confirm that standard and forward model checking require exponentially many iterations to verify the correct model while the translated version only needs a constant number of iterations in the correct case. All algorithms can find a counterexample with a constant number of iterations. Fig. 7.2 shows that both classical and forward model checking need time exponential in n. The translated variant can be checked in linear time. The standard algorithm is more than 25 % faster than forward model checking. A counterexample is found in the erroneous version in linear time by all algorithms. Performing forward model checking on the translated variant gives similar results as performing standard model checking.

7.2 Real-World Examples

In this section we report on a series of experiments with examples of nontrivial complexity.

Models Most models are taken from a collection of benchmarks [Yan] by Bwolen Yang, one is from the work of Latvala et al. [LBHJ05], and one is from previous work of the author [SB03]. For “1394” and “dme” we use instances of different sizes as indicated by the numerical parameters. Table 7.1 provides a brief description of the models.

Properties Templates of the properties used are given in Tab. 7.2. If a property was also used in [LBHJ05], it is referred to as “L”. The negated version of a property “p” is marked “¬ p”. One of the properties was made a liveness property by prefixing it with **F**. Other properties were made more interesting by requiring left sides of implications to hold infinitely often (marked “nv” for non-volatile). Some of the remaining properties had already been used in [SB04].

model	state bits	description	source
1394-3/4-2	97/137	IEEE 1394 FireWire tree identify protocol with 3 or 4 nodes and 2 ports per node	[SB03]
abp4	30	alternating bit protocol for 4 bits	[Yan]
bc57-sensors	78	reactor system model	[Yan]
brp	45	bounded retransmission protocol	[Yan]
dme5/6	90/108	asynchronous distributed mutual exclusion circuit with 5 or 6 nodes	[Yan]
pci	64	PCI Bus protocol	[Yan]
prod-cons	26	producer consumer	[Yan]
production-cell	54	production cell control model	[Yan]
srg5	8	5 bit shift register	[LBHJ05]

Table 7.1: Real-world examples: models

model	property	truth	pod	template
1394-3/4-2	0	t	1	$(\mathbf{F}(\mathbf{G}(p))) \rightarrow (\neg((q)\ \mathbf{S}\ (r)))$
	¬ 0	f	1	$\neg((\mathbf{F}(\mathbf{G}(p))) \rightarrow (\neg((q)\ \mathbf{S}\ (r))))$
	1	t	0	$\mathbf{F}((p) \vee ((q \vee (r)))$
	¬ 1	f	0	$\neg(\mathbf{F}((p) \vee ((q \vee (r))))$
	2	t	4	$\mathbf{G}((\mathbf{O}((p) \wedge (\mathbf{O}((\neg(p)) \wedge (\mathbf{O}((p) \wedge (\mathbf{O}(\neg(p))))))))) \rightarrow (\mathbf{F}(\mathbf{G}(\mathbf{X}(\neg(p))))))$
	3	t	6	$\mathbf{G}((\mathbf{O}((p) \wedge (\mathbf{O}((\neg(p)) \wedge (\mathbf{O}((p) \wedge (\mathbf{O}((\neg(p)) \wedge (\mathbf{O}((p) \wedge (\mathbf{O}(\neg(p))))))))))))) \rightarrow (\mathbf{F}(\mathbf{G}(\mathbf{X}(\neg(p))))))$
	4	t	8	$\mathbf{G}((\mathbf{O}((p) \wedge (\mathbf{O}((\neg(p)) \wedge (\mathbf{O}((p) \wedge (\mathbf{O}((\neg(p)) \wedge (\mathbf{O}((p) \wedge (\mathbf{O}((\neg(p)) \wedge (\mathbf{O}((p) \wedge (\mathbf{O}(\neg(p))))))))))))))))) \rightarrow (\mathbf{F}(\mathbf{G}(\mathbf{X}(\neg(p))))))$
abp4	0	t	0	$\mathbf{G}(\mathbf{F}(p))$
	L	f	2	$\mathbf{G}((p) \rightarrow (\mathbf{Y}(\mathbf{H}(q))))$
	¬ L	t	2	$\neg(\mathbf{G}((p) \rightarrow (\mathbf{Y}(\mathbf{H}(q)))))$
bc57-sensors	0	t	2	$\mathbf{G}(\mathbf{F}((p) \wedge (\mathbf{O}((q) \wedge (\mathbf{F}((r) \wedge (\mathbf{O}(s))))))))$
	¬ 0	f	2	$\neg(\mathbf{G}(\mathbf{F}((p) \wedge (\mathbf{O}((q) \wedge (\mathbf{F}((r) \wedge (\mathbf{O}(s)))))))))$
brp	L	t	2	$\mathbf{F}(\mathbf{G}((p) \rightarrow (\mathbf{O}((q) \rightarrow (O(r))))))$
	¬ L	f	2	$\neg(\mathbf{F}(\mathbf{G}((p) \rightarrow (\mathbf{O}((q) \rightarrow (O(r)))))))$
	¬ L, nv	f	2	$\neg((\mathbf{F}(\mathbf{G}((p) \rightarrow (\mathbf{O}((q) \rightarrow (\mathbf{O}(r))))))) \wedge ((\mathbf{G}(\mathbf{F}(p))) \wedge (\mathbf{G}(\mathbf{F}(q)))))$
dme5/6	L	f	2	$\mathbf{G}((p) \rightarrow ((p)\ \mathbf{T}\ ((\neg(p))\ \mathbf{T}\ (\neg(q)))))$
	¬ L	f	2	$\neg(\mathbf{G}((p) \rightarrow ((p)\ \mathbf{T}\ ((\neg(p))\ \mathbf{T}\ (\neg(q))))))$
	¬ L, nv	f	2	$\neg((\mathbf{G}((p) \rightarrow ((p)\ \mathbf{T}\ ((\neg(p))\ \mathbf{T}\ (\neg(q)))))) \wedge (\mathbf{G}(\mathbf{F}(p))))$
pci	L	f	4	$\mathbf{G}((p) \rightarrow (\mathbf{G}(((q) \wedge (\mathbf{Y}((r) \wedge (\mathbf{O}((s) \wedge (\mathbf{O}((t) \wedge (\mathbf{O}(u))))))))) \rightarrow (\mathbf{O}((v) \wedge (\mathbf{O}((w) \wedge (\neg(\mathbf{O}(x))))))))))$
	¬ L	f	4	$\neg(\mathbf{G}((p) \rightarrow (\mathbf{G}(((q) \wedge (\mathbf{Y}((r) \wedge (\mathbf{O}((s) \wedge (\mathbf{O}((t) \wedge (\mathbf{O}(u))))))))) \rightarrow (\mathbf{O}((v) \wedge (\mathbf{O}((w) \wedge (\neg(\mathbf{O}(x)))))))))))$
	$\mathbf{F}$ L	f	4	$\mathbf{F}(\mathbf{G}((p) \rightarrow (\mathbf{G}(((q) \wedge (\mathbf{Y}((r) \wedge (\mathbf{O}((s) \wedge (\mathbf{O}((t) \wedge (\mathbf{O}(u))))))))) \rightarrow (\mathbf{O}((v) \wedge (\mathbf{O}((w) \wedge (\neg(\mathbf{O}(x)))))))))))$
prod-cons	0	f	1	$((\mathbf{G}(\neg(p))) \wedge (\mathbf{G}(\mathbf{F}((q) \wedge ((q)\ \mathbf{S}\ (r)))))) \wedge (\mathbf{G}(\mathbf{F}(((q) \wedge ((q)\ \mathbf{S}\ (r))) \rightarrow ((s)\ \mathbf{S}\ (t)))))$
	¬ 0	f	1	$\neg(((\mathbf{G}(\neg(p))) \wedge (\mathbf{G}(\mathbf{F}((q) \wedge ((q)\ \mathbf{S}\ (r)))))) \wedge (\mathbf{G}(\mathbf{F}(((q) \wedge ((q)\ \mathbf{S}\ (r))) \rightarrow ((s)\ \mathbf{S}\ (t))))))$
	1	t	4	$\mathbf{G}((p) \rightarrow ((p)\ \mathbf{S}\ ((q)\ \mathbf{S}\ ((r)\ \mathbf{S}\ ((s)\ \mathbf{S}\ (t))))))$
	¬ 1, nv	f	4	$\neg((\mathbf{G}((p) \rightarrow ((p)\ \mathbf{S}\ ((q)\ \mathbf{S}\ ((r)\ \mathbf{S}\ ((s)\ \mathbf{S}\ (t))))))) \wedge (\mathbf{G}(\mathbf{F}(p))))$
	2	f	0	$\mathbf{G}((p) \rightarrow (\mathbf{F}(((q) \wedge (r)) \wedge (s))))$
	3	f	0	$\mathbf{G}((p) \rightarrow (\mathbf{F}(q)))$
	4	t	0	$\mathbf{G}((p) \rightarrow (\mathbf{F}(q)))$
production-cell	0	t	6	$\mathbf{G}(\mathbf{F}(((p) \vee (q)) \wedge (\mathbf{O}((r) \wedge (\mathbf{O}(((s) \vee (t)) \wedge (\mathbf{O}((u) \wedge (\mathbf{O}(((s) \vee (t)) \wedge (\mathbf{O}(((v) \vee (w)) \wedge (\mathbf{O}(x))))))))))))))$
	¬ 0	f	6	$\neg(\mathbf{G}(\mathbf{F}(((p) \vee (q)) \wedge (\mathbf{O}((r) \wedge (\mathbf{O}(((s) \vee (t)) \wedge (\mathbf{O}((u) \wedge (\mathbf{O}(((s) \vee (t)) \wedge (\mathbf{O}(((v) \vee (w)) \wedge (\mathbf{O}(x)))))))))))))))$
	1	t	12	$\mathbf{G}(\mathbf{F}(((p) \vee (q)) \wedge (\mathbf{Y}(\mathbf{O}((r) \wedge (\mathbf{Y}(\mathbf{O}(((s) \vee (t)) \wedge (\mathbf{Y}(\mathbf{O}((u) \wedge (\mathbf{Y}(\mathbf{O}(((s) \vee (t)) \wedge (\mathbf{Y}(\mathbf{O}(((v) \vee (w)) \wedge (\mathbf{Y}(\mathbf{O}(x))))))))))))))))))))$
	¬ 1	f	12	$\neg(\mathbf{G}(\mathbf{F}(((p) \vee (q)) \wedge (\mathbf{Y}(\mathbf{O}((r) \wedge (\mathbf{Y}(\mathbf{O}(((s) \vee (t)) \wedge (\mathbf{Y}(\mathbf{O}((u) \wedge (\mathbf{Y}(\mathbf{O}(((x) \vee (t)) \wedge (\mathbf{Y}(\mathbf{O}(((v) \vee (w)) \wedge (\mathbf{Y}(\mathbf{O}(x)))))))))))))))))))))$
	2	t	10	$\mathbf{G}(\mathbf{F}(((\neg(p)) \vee (\neg(q))) \wedge (\mathbf{O}((\neg(r)) \wedge (\mathbf{Y}(\mathbf{O}(((\neg(s)) \vee (\neg(t))) \wedge (\mathbf{O}((\neg(u)) \wedge (\mathbf{Y}(\mathbf{O}(((\neg(s)) \vee (\neg(t))) \wedge (\mathbf{Y}(\mathbf{O}(((\neg(v)) \vee (\neg(w))) \wedge (\mathbf{Y}(\mathbf{O}(x)))))))))))))))))$
	¬ 2	f	10	$\neg(\mathbf{G}(\mathbf{F}(((\neg(p)) \vee (\neg(q))) \wedge (\mathbf{O}((\neg(r)) \wedge (\mathbf{Y}(\mathbf{O}(((\neg(s)) \vee (\neg(t))) \wedge (\mathbf{O}((\neg(u)) \wedge (\mathbf{Y}(\mathbf{O}(((\neg(s)) \vee (\neg(t))) \wedge (\mathbf{Y}(\mathbf{O}(((\neg(v)) \vee (\neg(w))) \wedge (\mathbf{Y}(\mathbf{O}(x))))))))))))))))))$
srg5	L	t	4	$(((\mathbf{F}(\mathbf{G}(\neg(p)))) \wedge (\mathbf{G}(\mathbf{F}(q)))) \wedge (\mathbf{G}(\mathbf{F}(r)))) \rightarrow (\mathbf{F}((s)\ \mathbf{S}\ ((t)\ \mathbf{S}\ ((u)\ \mathbf{S}\ ((v)\ \mathbf{S}\ (w))))))$
	¬ L	f	4	$\neg((((\mathbf{F}(\mathbf{G}(\neg(p)))) \wedge (\mathbf{G}(\mathbf{F}(q)))) \wedge (\mathbf{G}(\mathbf{F}(r)))) \rightarrow (\mathbf{F}((s)\ \mathbf{S}\ ((t)\ \mathbf{S}\ ((u)\ \mathbf{S}\ ((v)\ \mathbf{S}\ (w)))))))$
	¬ L, nv	f	4	$\neg(((((\mathbf{F}(\mathbf{G}(\neg(p)))) \wedge (\mathbf{G}(\mathbf{F}(q)))) \wedge (\mathbf{G}(\mathbf{F}(r)))) \rightarrow (\mathbf{F}((s)\ \mathbf{S}\ ((t)\ \mathbf{S}\ ((u)\ \mathbf{S}\ ((v)\ \mathbf{S}\ (w))))))) \wedge (((\mathbf{F}(\mathbf{G}(\neg(p)))) \wedge (\mathbf{G}(\mathbf{F}(q)))) \wedge (\mathbf{G}(\mathbf{F}(r)))))$

Table 7.2: Real-world examples: templates of the properties

Platform and resource bounds All experiments were performed on an Intel Pentium IV at 2.8 GHz with 2 GB RAM running Linux 2.4.18. We used NuSMV 2.2.2 [CCG+02, NuS] and Cadence SMV build 10-11-02p46 [McM] as model checkers for all experiments except for the comparison between SAT- and BDD-based methods to find shortest counterexamples. There, bounded model checking was performed with build 050429-CAV-final of a modified NuSMV 2.2.3, which implements an incremental version of the encoding of [LBHJ05]. It was presented in [HJL05] and is available from [HJL]. As SAT solvers we used zChaff 2004.11.15 [ZMMM01, zCh] and MiniSat 1.12 [ES04, ES]. In that comparison, model checking of the state-recording translation was performed with the corresponding unmodified version of NuSMV 2.2.3. Time and memory usage were limited to one hour and 1.5 GB for each run in all experiments.

Algorithms We use three different algorithms for fair cycle detection:

- **L2S** This denotes the state-recording translation as described in Chap. 3 with BDD-based invariant checking.
- **Live** This is the standard approach to perform LTL model checking in NuSMV [CCGR00]: BDD-based symbolic model checking for CTL [BCM+92] is employed to check $\neg\mathbf{EG}1$ under fairness.
- **BMC** This is incremental SAT-based bounded model checking [HJL05].

Encoding of the property We use an optimized encoding of the tight Büchi automaton presented in Chap. 5 to encode the property with “L2S”. The level of virtual unrolling can be chosen between no unrolling and full virtual unrolling to the past operator depth of the formula. The encoding is tightly integrated with the state-recording translation. As an example, the signals indicating the state of the loop are provided directly by the reduction rather than being separate input variables. Fairness is handled similar to to [LBHJ05]. In fact, the implementation started as an adaptation of the BMC encoding in [LBHJ05]. Only then a tight Büchi automaton was extracted from the construction. The original implementation was kept for (slightly) superior performance. The specification is given as `INVARSPEC`$\neg lc$ with NuSMV and as $\mathbf{AG}\neg lc$ with Cadence SMV.

For “Live” with a non-tight Büchi automaton we use NuSMV’s ltl2smv tool, which implements the encoding of Kesten et al. [KPR98]. Ltl2smv is invoked either explicitly (for Cadence SMV and for NuSMV in the comparison of a tight and a non-tight Büchi automaton with “Live”) or implicitly as part of the operation of NuSMV when `LTLSPEC` is used (for the comparison of “L2S” and “Live”). For “Live” with a tight Büchi automaton we use our own implementation of the translation from PLTLB to a tight Büchi automaton from Sect. 5.3.

For “BMC” the built-in encoding [HJL05] of the modified NuSMV is used.

Degrees of tightness We distinguish between the following degrees of tightness:

- **tight** stands for virtual unrolling up to the past operator depth of the formula. An example is the tight Büchi automaton from Sect. 5.3.

- **not tight** stands for no virtual unrolling at all. An example is a Büchi automaton constructed with [KPR98].

- **maxunroll**n stands for partial unrolling: each occurrence of $h_p(\psi)$ is replaced with $\min(h_p(\psi), n)$ in Tab. 5.1 (see also Sect. 5.3.3).

Variants of variable optimization We use the following variants of variable optimization:

- **none** means no variable optimization,

- **ic** combines removing constants after initialization and transition input variables (Thms. 34, 35),

- **ic(none)** stands for “ic” if the set of constant and transition input variables is not empty, for “none” otherwise,

- **coi** denotes including only variables in the fair cone of influence that are neither constant nor transition input variables (Thm. 37), and

- **absref** is abstraction refinement on the set of variables used in loop detection (Thm. 38).

Settings If a model-specific variable order is provided with the model, it is used for all experiments with “L2S” and “Live” except in the comparison between “L2S” and “BMC”. Original state variables and the copies arising in the state-recording translation are always interleaved when “L2S” is used. Dynamic reordering of variables and cone of influence reduction are disabled. Restriction to the reachable set of states is enabled explicitly in all cases but “L2S” with NuSMV (where this is assumed to be part of the algorithm employed for `INVARSPEC` [CCO+]). Checking for completeness is disabled in “BMC”.

Presentation of results We use scatter plots to compare the time or memory usage of two approaches a and b, where a on the x-axis corresponds to the “new” and b on the y-axis to the “standard” approach, respectively. Both axes have a logarithmic scale. An instance of the problem is solved with both approaches and a data point is obtained by taking the tuple (resource usage with a, resource usage with b). Smaller values, which are closer to the origin, are better. Hence, a data point *above* the $y = x$-diagonal indicates an advantage for the “new” approach a in that particular instance. Experiments where the property holds are marked with a filled green square, those where it is false with an empty red triangle. Three special values denote time out (“to”), memory out (“mo”), and other errors (“er”). In all instances of the latter Cadence SMV reports problems with a file that could not be resolved.

When comparing lengths of counterexamples we use bar charts. The “new” approach is the left red bar, while the “standard” approach is the right green bar. The results are sorted in ascending order w.r.t. the “standard” approach. Shorter lengths are better.

Raw data is provided in Appendix B.

7.2.1 State-Recording Translation versus Standard Approach

In this subsection we compare the performance of forward reachability checking with the state-recording translation ("L2S") to that of the standard method of checking linear properties [BCM+92, CGH97] ("Live") in BDD-based symbolic model checkers. A non-tight encoding of the property is used and no restriction on the past operator depth of the formula is applied.

Figure 7.3 shows the results. The left column is for NuSMV, the right column for Cadence SMV. From top to bottom the rows are CPU time (in seconds), memory usage (in BDD nodes), and, for false properties, length of the resulting counterexamples (in states).

Neither method has a clear advantage in terms of CPU time if the property under consideration is false: "L2S" is faster in about two thirds of the experiments with NuSMV while "Live" is faster in more cases with Cadence SMV. If the property is true, it can almost always be verified faster with the standard approach. In most cases "Live" uses less memory than "L2S". Counterexamples obtained with "L2S" are substantially shorter than those obtained with "Live".

7.2.2 BDD- versus SAT-based Model Checking of the Tight Encoding

In this subsection we investigate how finding shortest counterexamples with BDD-based symbolic model checking and the state-recording translation ("L2S") compares to the SAT-based bounded model checking variant that inspired our tight Büchi automaton [LBHJ05] ("BMC"). As the implementation of [LBHJ05] is based on NuSMV, Cadence SMV is not used in this set of experiments. Only properties that proved false are included and only the tight encoding is used. Our set of models and properties includes the ones from [LBHJ05].

The results are plotted in Fig. 7.4. The left column is for zChaff, the right is for MiniSat. CPU time is shown in the upper row (in seconds), memory usage in the lower row (in bytes).

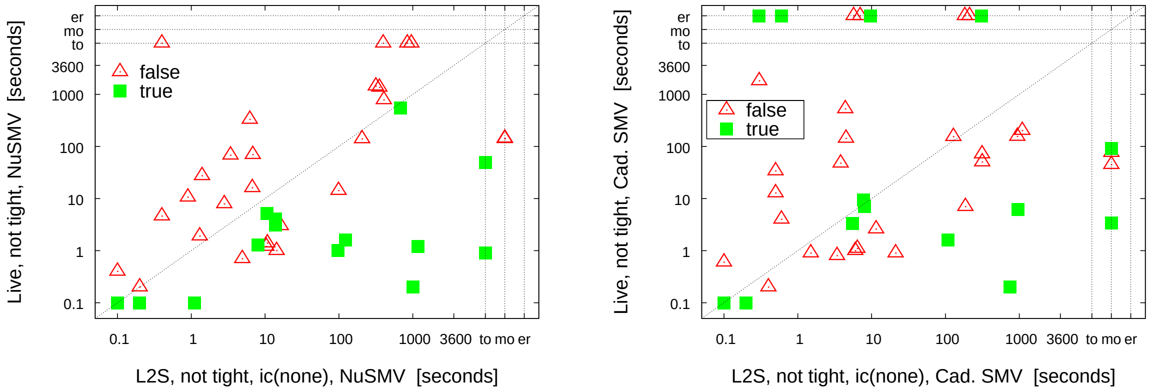
Although slightly more examples are solved faster with "BMC", neither algorithm has a clear advantage. Each algorithm outperforms the other by more than an order of magnitude for some examples. With respect to memory usage SAT-based bounded model checking is the better choice in most cases. [LBHJ05] also detects shortest informative counterexamples to safety properties [KV01], which can be shorter than the shortest lasso-shaped counterexample. Such a counterexample is reported for "pci,¬ L", but it is not shorter than the one found with "L2S".

7.2.3 The Cost of Tightness

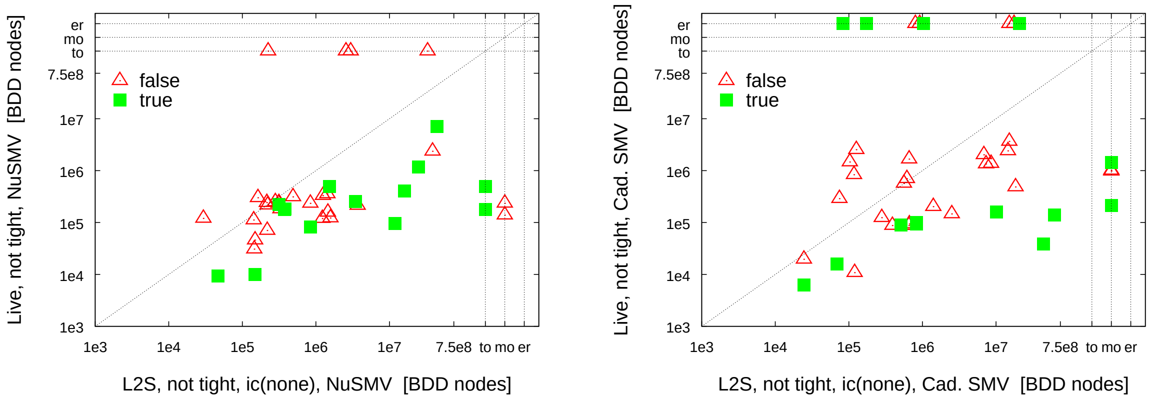
In this subsection we determine the impact of using encodings with different degrees of tightness on performance and on length of counterexamples.

Figure 7.5 shows the results. Again, NuSMV is on the left and Cadence SMV on the right, with CPU time in the top, memory usage in the middle, and length of counterexamples in the bottom row. We include only experiments with past operator depth at least 1 and where a result was obtained within the resource bounds for at least one degree of tightness. Moreover, we omit results that are identical to "tight", i.e., results for "maxunroll1" are shown only for $h_p(\phi) > 1$, and for "maxunroll2" only for $h_p(\phi) > 2$. Resource usage for each degree of tightness is depicted as the ratio* (i.e., the speed-up or slow-down) between the CPU time or memory usage

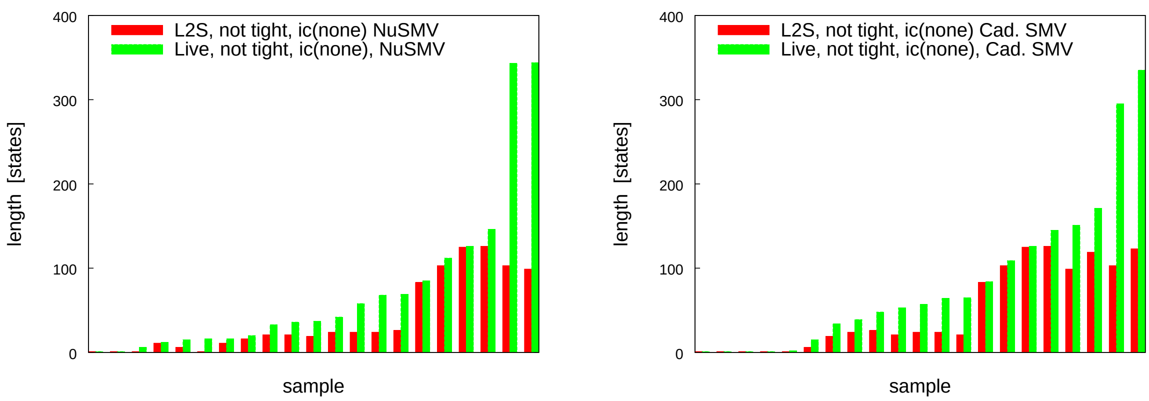
*For the absolute values see App. B



CPU time



Memory usage



Length of counterexamples

Figure 7.3: State-recording translation ("L2S") versus standard approach ("Live")

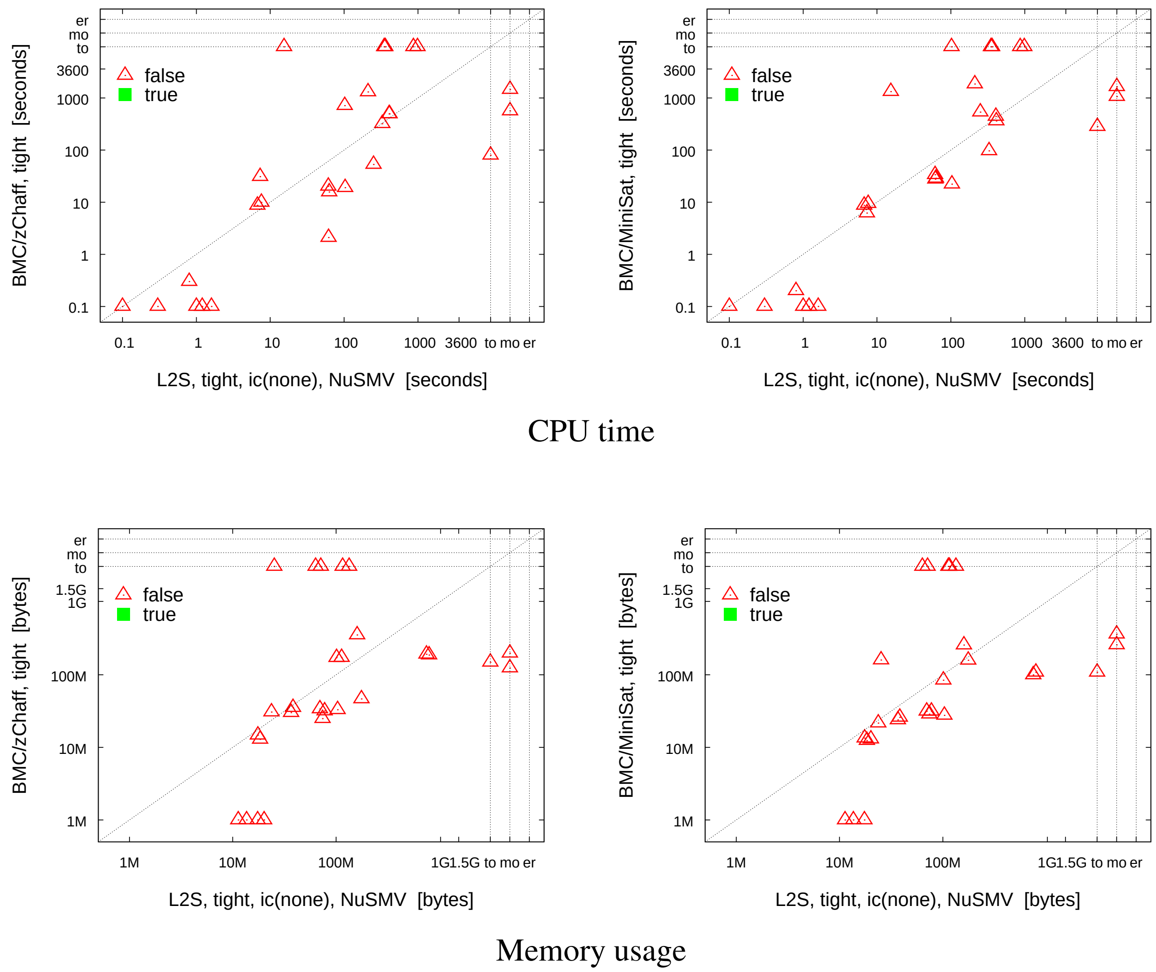


Figure 7.4: Finding shortest counterexamples using BDDs and the state-recording translation ("L2S") versus using SAT-based bounded model checking ("BMC")

of that degree and the corresponding value of the non-tight case on a logarithmic scale. Hence, the results for "not tight" are all 1. They are marked by the straight red line. The other (pink) line represents the results for "tight"; filled green triangles mark results for "maxunroll1", empty blue squares for "maxunroll2". The results are sorted in ascending order of the ratio between "tight" and "not tight".

While some examples can be solved using less time or memory with a tight encoding, for more than half of the cases tightness comes at a cost. A shorter counterexample is obtained for three combinations of a model and a property, in one of these the reduction is substantial. Raw data shows no correlation between a reduction in resource usage and a reduction in length of a counterexample. Compared to full tightness, limiting the maximum level of unrolling leads to more well-behaved resource usage and — for our examples — counterexamples of the same length. The intuition that more virtual unrolling tends to incur higher resource usage is (weakly) supported by our results.

7.2.4 Comparing Variants of Variable Optimization

We now compare the impact of the different variants of variable optimization.

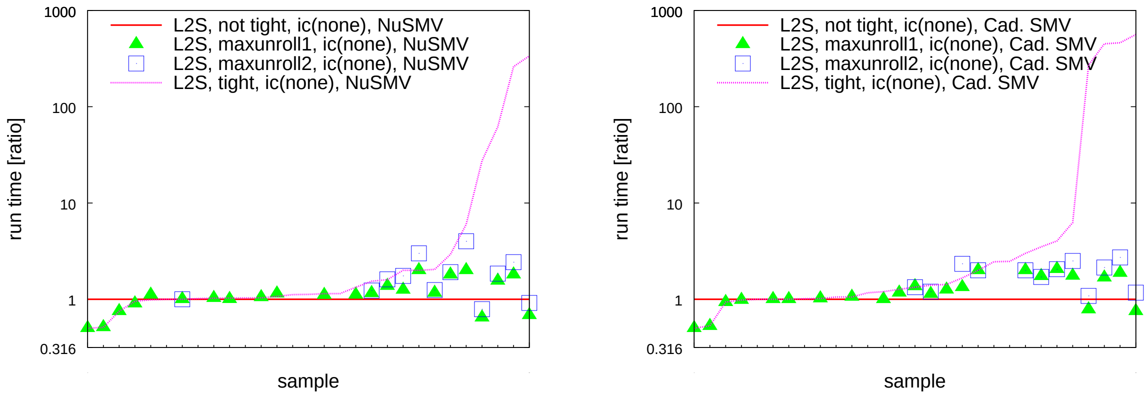
Figures 7.6 and 7.7 show time and memory usage, respectively. Results for false and true properties are shown separately in the left and right columns. Top and bottom rows correspond to NuSMV and Cadence SMV. Both, tight and non-tight encodings are included. Only experiments where a result was obtained within time and memory bounds for at least one variant are plotted. The results are depicted in a similar way as in the previous subsection: all results are shown as the ratio (speed-up/slow-down) between CPU time or memory usage of one variant and that of "none" using a logarithmic scale. Hence, the straight red line corresponds to "none", the pink line to "absref", the green filled triangles to "ic", and the empty blue squares to "coi". The results are sorted in ascending order of the ratio between "ic" and "none".

The results show that "ic" almost always leads to lower resource usage. The few exceptions where higher resource usage is incurred are all examples with less than 2 seconds run time. Reducing the set of variables in loop detection from "ic" to "coi" leads to a further reduction in time and memory consumption in about half of the cases. There seem to be no major differences between true and false properties for "ic" and "coi". That changes for "absref". If the property is false, and when compared to "none", "absref" is helpful more often than not for run time and memory usage with Cadence SMV and for memory usage with NuSMV. However, compared to "coi", there is hardly a benefit and often a penalty. If, on the other hand, the property is true, "absref" often leads to significant speed-up and reduced memory consumption. In a number of cases a property can be proven to be true in a few seconds using "absref" although it timed or mem'ed out with "none", "ic", and "coi". Still, "absref" does not make BDD-based verification of a true property faster with "L2S" than with "Live", even if no abstraction at all is applied in the "Live" case.

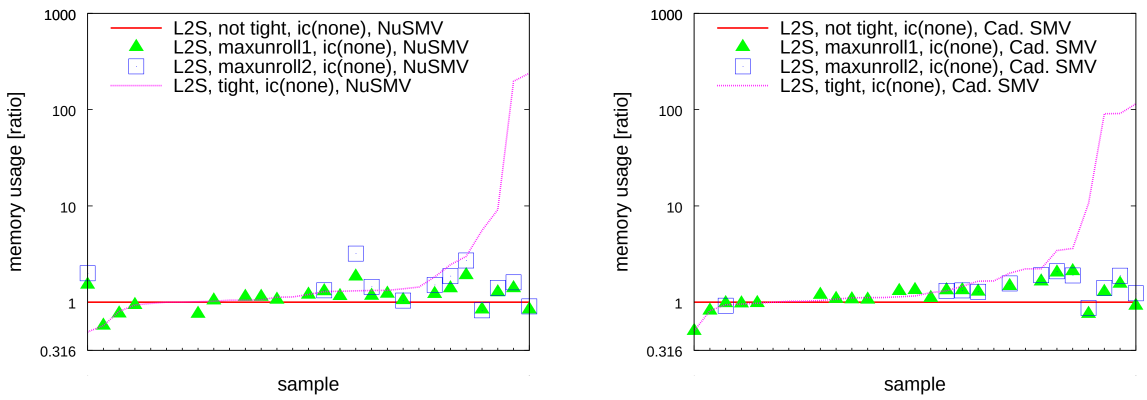
7.2.5 A Tight Büchi Automaton in the Standard Approach

The last set of experiments examines whether it is beneficial to use a tight Büchi automaton in the standard approach to LTL model checking.

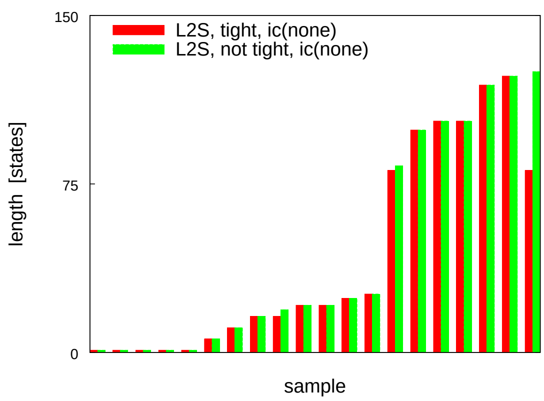
The results are shown in Fig. 7.8. The order of diagrams is the same as for "L2S" versus



CPU time



Memory usage



Length of counterexample

Figure 7.5: Comparing degrees of tightness

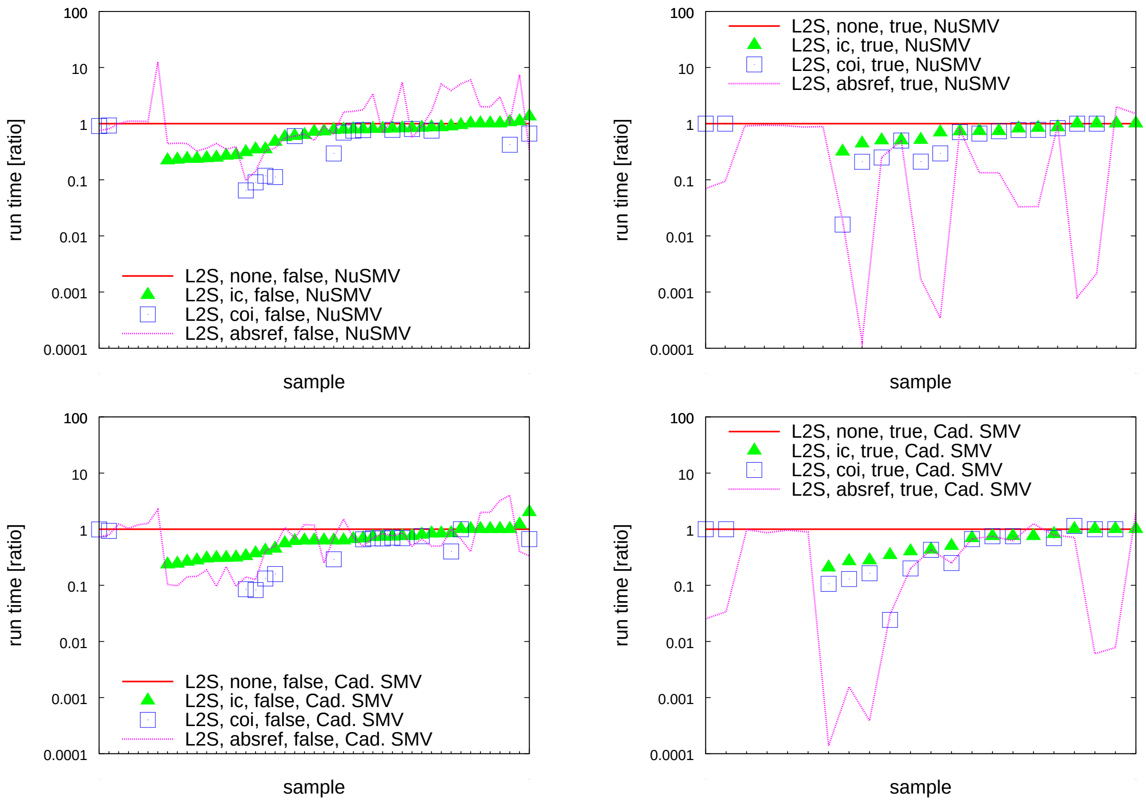


Figure 7.6: Comparing degrees of variable optimization: CPU time

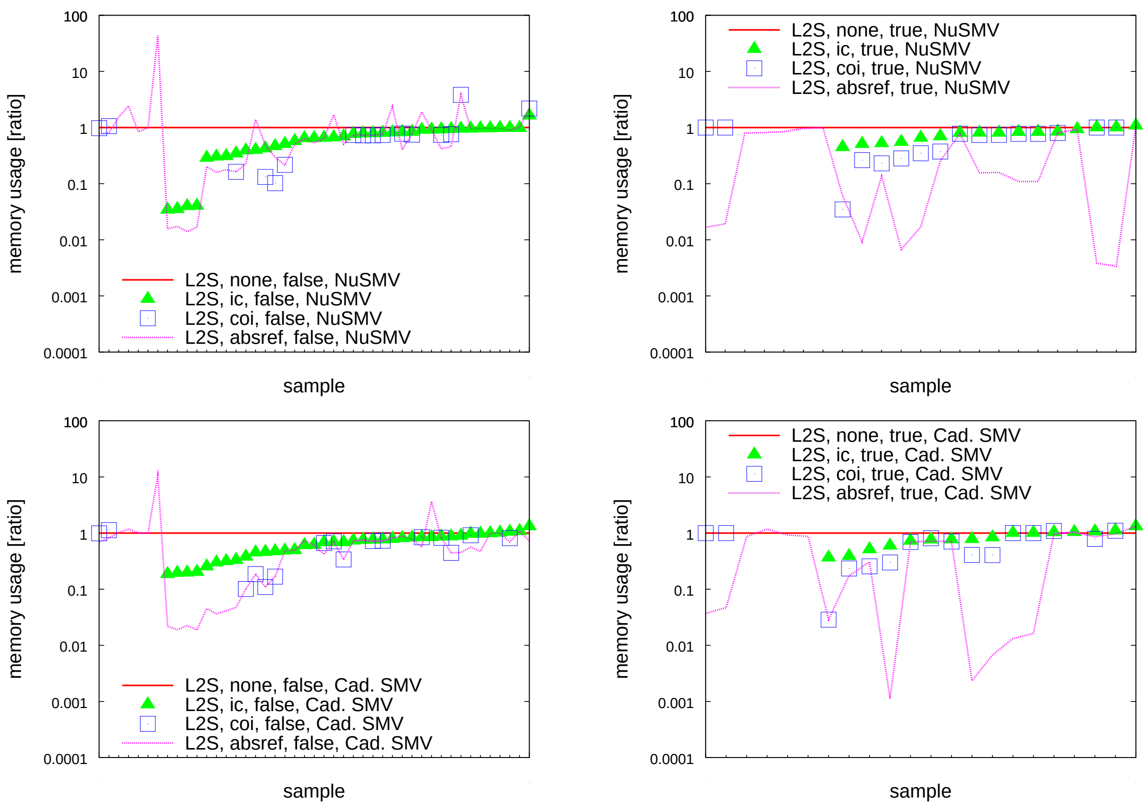


Figure 7.7: Comparing degrees of variable optimization: memory usage

"Live": NuSMV in the left column, Cadence SMV in the right column, CPU time (in seconds) in the top row, memory usage (in BDD nodes) in the middle row, and length of the resulting counterexamples (in states) if the property is false.

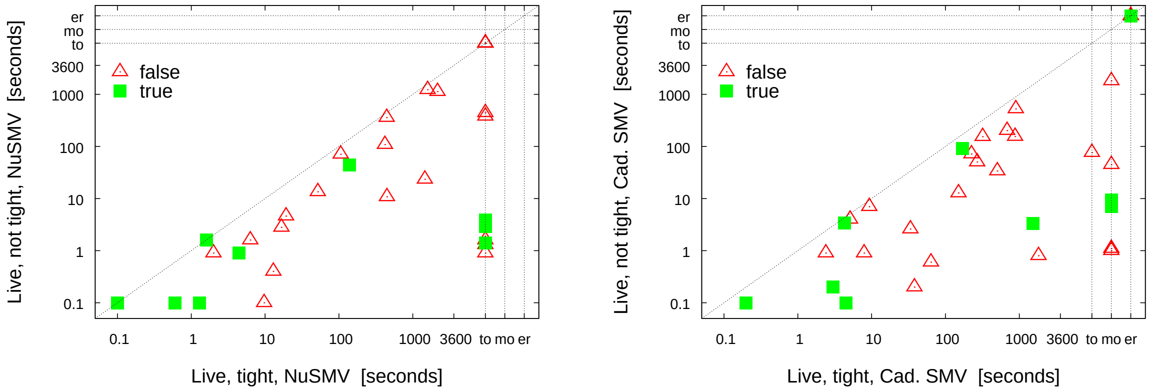
The results show that more often than not *longer* counterexamples are reported by both, NuSMV and Cadence SMV. Making matters worse, significantly more CPU time is spent and memory is needed with a tight automaton for almost all examples.

7.3 Summary

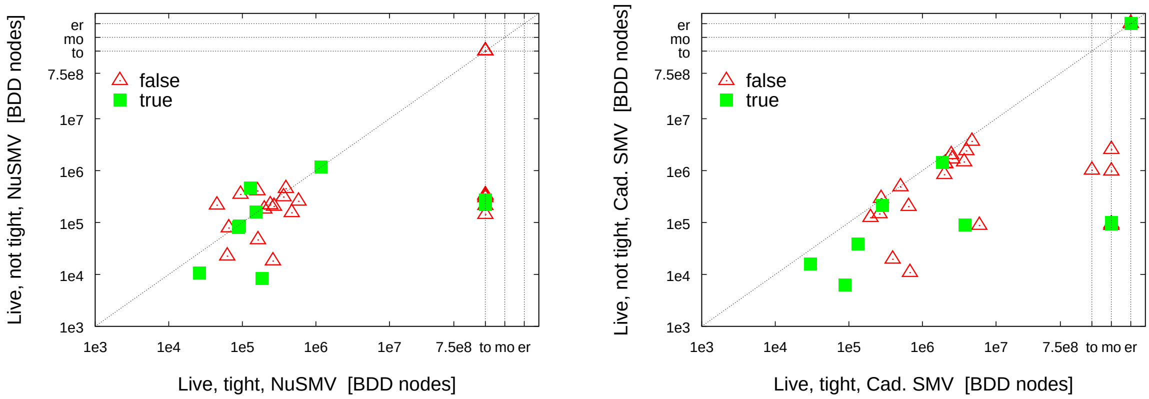
Although the state-recording translation doubles the number of state variables in the worst case, on many practical examples the overhead of the translation turns out to be quite reasonable. Falsification of a property often takes less time and memory when the state-recording translation is used. A simple liveness property of a forward-jumping counter can even be verified exponentially faster with the state-recording translation than with a traditional symbolic model checking algorithm for liveness [BCM+92].

Experimental results confirm that the performance of the state-recording translation in BDD-based symbolic model checking depends to a large part on the set of variables included in loop detection. Reducing that set benefits both run time and memory usage. Performing abstraction refinement on the set of variables used for loop detection can improve performance by more than 2 orders of magnitude for passing properties, while it incurs a penalty for failing properties. Similar observations on the effect of abstraction refinement have been made by, e.g., [BGG02, LS06].

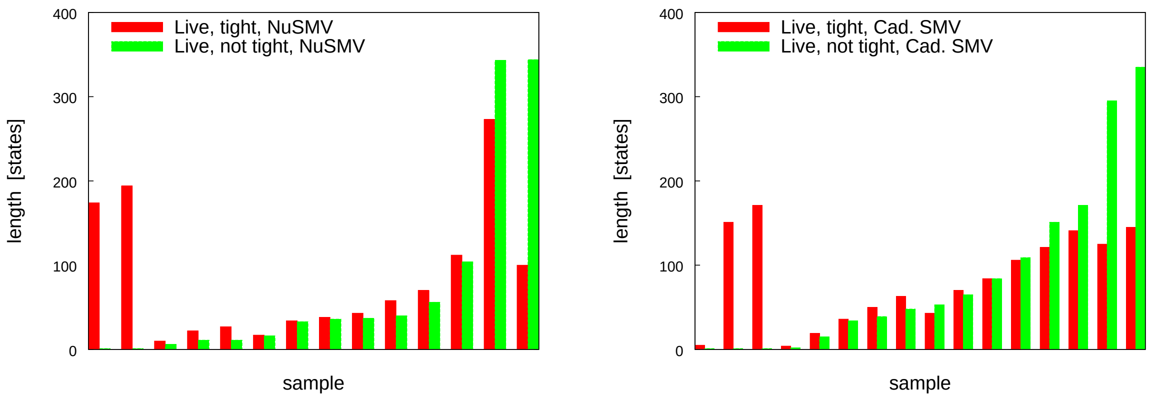
The standard algorithm for finding a fair cycle in symbolic model checking [CGMZ95] typically does not produce a shortest fair cycle. That can be found by applying the state-recording translation and performing a breadth-first reachability check. Our experimental results show a substantial reduction in the length of counterexamples to linear time properties when the latter approach is employed. The additional step from the (non-tight) automaton [KPR98] to the tight Büchi automaton from Sect. 5.3 only leads to a marginal further improvement; using a tight automaton comes at a cost in terms of run time and memory consumption. For that reason Cimatti et al. did not implement virtual unrolling in their approach [CRS04] to bounded model checking [Cim05]. While the benefit of a tight automaton may not show up on all examples, we believe that the user should have the option to decide whether shortest counterexamples are desirable. Limiting the amount of virtual unrolling between no and full unrolling allows a user to trade an increase in resource usage for a decrease in counterexample length. Using the tight automaton with a standard model checking algorithm [BCM+92] in many cases results in longer counterexamples and increased resource usage. The combination of breadth-first reachability checking of the state-recording translation and an optimized implementation of the tight Büchi automaton gives a BDD-based method to find shortest counterexamples that is competitive with a SAT-based bounded model checker [HJL05] for that purpose.



CPU time



Memory usage



Length of counterexamples

Figure 7.8: Tight versus not tight automaton with “Live”

8 Conclusion

Paradoxically, $\mathbf{F}\,dead$ *is a liveness property.*
It even appears to be true for animals and humans...
Markus Müller-Olm, Liveness Manifesto [PSZ]

8.1 Contributions

The distinction between liveness and safety properties is still fundamental in verification. Proving safety and liveness properties are different problems with different decidability results, algorithms, tools, and papers. Safety properties are considered more important in practice than liveness properties. In addition, performance issues can prevent liveness properties from being verified on models of equal size or level of detail as the corresponding safety properties. Still, proving termination remains an important task. In a workshop on verification "Beyond Safety", Andreas Podelski remarked [Pod04]

> The issue "liveness less useful than safety" becomes obsolete once we have shown that the good methods for checking safety are, in fact, methods for checking liveness (we are working on it).

This dissertation is a step in that direction.

State-recording translation We presented a reduction from checking fair repeated reachability to checking reachability for finite state systems. The so-called state-recording translation extends a finite state system such that it can nondeterministically save a copy of the current state of the original state variables. It then waits until all fairness constraints have been met. When this is the case, and a second occurrence of the saved state is seen, an initialized fair lasso-shaped path, i.e., a counterexample to the property, has been found. The translation leads to a quadratic blowup in the size of the state space and a small, constant increase in the radius and diameter of the system. It is applicable to all ω-regular properties.

Optimizations Two optimizations proved vital to make the state-recording translation work in practice. First, BDD variables representing original state variables must be interleaved with their copies introduced by the state-recording translation. Second, variable optimization considers only a subset of the state variables for loop detection. Experimental results confirm that the overhead of the translation depends to a large part on the variables included in loop detection.

Extension to infinite state systems We extended the translation to some classes of infinite state systems, namely, regular model checking [KMM+01, WB98, BJNT00], pushdown systems [BEM97, FWW97, EHRS00a], and timed automata [AD94]. In all cases the reduction expresses an existing algorithm for liveness checking of this class of systems syntactically in this class: it "pulls the algorithm into the model."

Experimental evaluation: overhead of state-recording translation We conducted a number of experiments with finite state systems, which show that the transformed system can be verified with acceptable overhead in practice. To our pleasant surprise, for some systems even a considerable speed-up can be obtained. We gave an example where the speed-up is exponential.

Finding shortest counterexamples Counterexamples are a salient feature of model checking. They help users to find errors and, more recently, are also used as part of model checking algorithms or to locate errors automatically. If a counterexample is to be interpreted by a human (and most still are), it should provide (only) information guiding the user to the error. Hence, short counterexamples can be helpful. If breadth-first reachability checking is performed on the transformed system, shortest fair cycles are obtained. In the automaton-based approach to model checking this cycle is a cycle in the product of the model and a Büchi automaton accepting counterexamples to the specification. Hence, the Büchi automaton for the specification must be such that it does not prevent a shortest counterexample in the model to be found.

Extending tightness to Büchi automata A finite automaton on finite words is tight if it accepts shortest prefixes to safety properties [KV01]. We extended that notion to Büchi automata and provided necessary and sufficient criteria for tightness. We proved that frequently used translations from LTL with (Kesten et al. [KPR98]) or without (Gerth et al. [GPVW96]) past to Büchi automata do not yield tight Büchi automata. We showed that resulting counterexamples may have excess length linear in the length of the specification. We adapted a construction [LBHJ05, BC03] from bounded model checking to Büchi automata in order to get a translation from full LTL or from an arbitrary Büchi automaton into a tight Büchi automaton.

Experimental evaluation: finding shortest counterexamples We combined the tight Büchi automaton with the state-recording translation and obtained a practical algorithm to find shortest counterexamples to linear time properties with a BDD-based symbolic model checker. Our experimental results indicate competitive performance with SAT-based bounded model checking [HJL05]. They also show a clear benefit of using a model checking algorithm that finds shortest cycles: counterexamples produced by using breadth-first search in the transformed model are on average one third shorter than counterexamples produced by the traditional algorithm [CGMZ95] in the original model. The benefit of using a tight automaton for full LTL proved negligible.

8.2 Future Work

Extending application areas The state-recording translation has already been picked up by a number of researchers. McMillan uses it to verify liveness properties with his interpolation-based approach [McM03]. Claessen also reports good performance of that combination [Cla06]. Edelkamp and Jabbar apply the translation in the context of directed, external, and distributed explicit-state model checking [EJ06]. Preliminary experiments [BHJ+06] indicate problematic performance with induction-based bounded model checking [ES03], which may be due to a high degree of non-determinism w.r.t. the encoding of the property in our current implementation [Hel06]. Other potential application areas include runtime monitoring and testing. In runtime monitoring it might be useful to store more than one potential loop start. Shortest counterexamples could also benefit abstraction refinement schemes that need to reconstruct counterexamples as the reconstruction itself might explode [BGG02]. Short counterexamples might also be of interest in the domain of planning.

8.2.1 State-Recording Translation

Powerful yet efficient reductions Reductions from logics more powerful than PLTLB to reachability have been suggested [SYE+05]. However, the methods of [SYE+05] incur a high penalty in performance, which puts their practical application largely out of reach. One direction for future work is therefore to try increasing the power of the state-recording translation while retaining its (relative) efficiency. For that, a theoretical framework to characterize and classify reductions from one logic to another, where the reductions are allowed to modify both model and formula, could prove to be a helpful tool. The notion of "model checking power" introduced by Shilov et al. [SYE+05] is a potential starting point. Another point for consideration is the fact that we pulled an existing algorithm for verifying liveness properties into the model in all our examples.

Optimization We have mostly focused on variable optimization as it directly targets the source of the overhead of the state-recording translation. A number of other optimizations come to mind. The state-recording translation can be combined with the counter-based approach by limiting the search depth of the breadth-first search using any bound that guarantees that a potential shortest counterexample is preserved. Empirical evidence suggests that many practical systems have a relatively small radius, which can be computed using a structural algorithm [BKA02]. However, a practical method to derive small bounds on the length of a shortest counterexample for ω-regular properties is not yet available. Our current implementation may guess a loop start at any state. Heuristics, which limit the set of potential loop starts, could also help to reduce the overhead of the translation. In an explicit state setting it might be useful to guess a loop start only if a state is initial or has at least two incoming transitions.* Another restriction (applied, e.g., in distributed explicit state model checking [BBČ02]) might also be useful in a symbolic setting: no loop start should be guessed when the system has not yet entered a fair strongly connected component. Computing the latter for the product of the model, M, and the Büchi automaton representing the property, B, is clearly equivalent to solving the original model checking problem; for smaller properties, e.g., when searching for a witness of $\mathbf{FG}p$, the computation should be feasible for B. This could give some of benefits of the nested depth-first search algorithm in explicit state model checking [CVWY92], which only starts the nested search in an accepting state, while retaining shortest counterexamples.

*This suggestion was made by a participant at CAV/ISSTA 2004 in Boston whose name I unfortunately forgot.

Criteria when state-recording improves performance We currently don't have criteria, which allow to determine a priori whether a model can be model checked faster with or without state-recording translation applied. Given that performance between both methods may vary by more than an order of magnitude such criteria are desirable.

Experimental comparison with bounded liveness checking Finally, it would be interesting to perform an experimental comparison of the performance of bounded liveness checking with optimal bounds and the state-recording translation.

8.2.2 Infinite State Systems

Criteria for infinite state systems For infinite state systems more general criteria for the applicability of the state-recording translation should be sought. Considering examples where liveness is undecidable and seeing why the state-recording translation cannot be used could help to understand the limits of the transformation.

Termination in regular model checking Regular model checking is undecidable in general [AK86]. Clearly, we cannot expect that computing the transitive closure of the transition relation terminates on the transformed model when it doesn't on the original model. However, it is an open question whether that computation terminates on the transformed model in all cases in which it does on the original model.

Reducing overhead for timed automata Clock zones [Alu99] have been very helpful to increase performance of model checking timed automata. They might, therefore, also help to limit the overhead of the state-recording translation for timed automata.

Experimental evaluation Finally, only limited experimental evaluation of the state-recording translation has been performed for infinite state systems so far [BHV04].

8.2.3 Tight Büchi Automata

Experimental evaluation of excess length of [GPVW96] Due to their size, both the tight automaton for PLTLF [KPR98] and our tight automaton for PLTLB are not well-suited for explicit-state model checking. Our results indicate that the step from [KPR98] to an automaton that is tight for full LTL has limited benefits. We are not aware of a corresponding empirical study that determines the excess length of counterexamples produced by automata such as [GO01], which are preferred in an explicit setting, compared to those produced by [KPR98] or by the tight automaton from Sect. 5.3. Such a study could help to decide whether it is worthwhile to come up with tight(er) automata for explicit state model checking.

Small tight automata If it turns out that small tight automata are desirable, we should try to understand precisely which features of the construction of [GPVW96] prevent it from accepting shortest counterexamples. An initial observation is that [GPVW96] follows a lazy approach in evaluating the truth of subformulae: it only tracks the truth of those subformulae, which it deems necessary to establish truth of the specification. In contrast, [KPR98] tracks the truth of all subformulae simultaneously. Finding middle ground here could be a first step to a tight(er) automaton than [GPVW96]. Another open question is how the various optimizations, which have been suggested for [GPVW96], influence tightness of an automaton.

Tightness for true properties Tightness of a Büchi automaton representing the property to be verified ensures that the encoding of the property does not hurt the length of potential counterexamples. The complementary property seems also desirable: if the property turns out to be true, the encoding of a property as a Büchi automaton should not lead to larger termination depths than required by the model to be verified. Empirical evidence shows that [KPR98] leads to higher termination depths than [SB00] in bounded model checking [AS06]. We also have preliminary theoretical results that virtual unrolling doesn't lead to smaller termination depths in the bounded model checking approach of Heljanko et al. [HJL05].

Missing theory Given the emergence of specification languages such as PSL [Acc], efficient translations from extended linear temporal logics [Wol83, SVW87, VW94] to tight Büchi automata would be interesting. We also lack a lower bound on the size of a tight automaton for PLTLB. Further, it is unclear how our results transfer to other classes of automata, be it Muller, Rabin, or Streett automata on infinite words or automata on infinite trees. Among the basic facts that we have established for tight automata is preservation of tightness under language union and intersection using Büchi automata. We have not looked yet at complementation. Finally, Chap. 5 gives two procedures to make an arbitrary Büchi automaton tight. Nothing is known about their relative merits.

Tightness + ? Another area for future research is the combination of tightness with other approaches that aim to make counterexamples easier to understand [JRS02, RS04, GK05]. The ideas of obtaining "nice" values from a SAT solver [GK05] and of fate and free will in error traces [JRS02] seem orthogonal to obtaining shortest counterexamples. However, if complex state changes in the model are split into atomic steps that change only one or few state variables, the shortest counterexample has a high likelihood to also exhibit the fewest changes of state variables. Clearly, there might be a negative impact on the number of reachable states in the model.

Optimization We have not yet extensively optimized the encoding of a tight automaton for the state-recording translation. Recent work in bounded model checking [LBHJ05, HJL05] should be used to identify opportunities.

A

Proofs and Auxiliary Lemmas

Lemma 1 *Let $\langle\beta,\gamma\rangle$ be a minimal lasso for α, $\langle\beta',\gamma'\rangle$ a minimal lasso for α', and $\alpha'' = \alpha\times\alpha'$. Then there are finite sequences β'',γ'' such that $\langle\beta'',\gamma''\rangle$ is a minimal lasso for α'', $|\beta''| = \max(|\beta|,|\beta'|)$, and $|\gamma''| = lcm(|\gamma|,|\gamma'|)$.*

Proof: Let $l_s = \max(|\beta|,|\beta'|)$, $l_l = lcm(|\gamma|,|\gamma'|)$. Define $\beta'' = \alpha''(0)\ldots\alpha''(l_s-1)$, $\gamma'' = \alpha''(l_s)\ldots\alpha''(l_s+l_l-1)$. Clearly, $\alpha'' = \beta''\gamma''^{\omega}$. Now assume, there exist $\hat{\beta}'',\hat{\gamma}''$, such that $\alpha'' = \langle\hat{\beta}'',\hat{\gamma}''\rangle$ and $|\langle\hat{\beta}'',\hat{\gamma}''\rangle| < |\langle\beta'',\gamma''\rangle|$.

Assume first that $|\hat{\beta}''| < |\beta''|$. W.l.o.g. $|\beta'| \leq |\beta|$. By projecting $\hat{\beta}'',\hat{\gamma}''$ onto their first components we can extract $\hat{\beta},\hat{\gamma}$ with $\alpha = \hat{\beta}\hat{\gamma}^{\omega}$ and $|\hat{\beta}| < |\beta|$. Further, there exists $0 \leq i < |\gamma|$ such that $\gamma_{rot} = \gamma(i)\ldots\gamma(|\gamma|-1)\gamma(0)\ldots\gamma(i-1)$ with $|\gamma_{rot}| = |\gamma|$ and ${\gamma_{rot}}^{\omega} = \hat{\gamma}^{\omega}$. Hence, $\langle\hat{\beta},\gamma_{rot}\rangle = \langle\beta,\gamma\rangle$ with $|\langle\hat{\beta},\gamma_{rot}\rangle| < |\langle\beta,\gamma\rangle|$, a contradiction.

Now assume $|\hat{\gamma}''| < |\gamma''|$. W.l.o.g. $|\gamma|$ does not divide $|\hat{\gamma}''|$. By projecting $\hat{\gamma}''$ onto its first component we can extract $\hat{\gamma}$ with $\gamma^{\omega} = \hat{\gamma}^{\omega}$ and $|\gamma|$ does not divide $|\hat{\gamma}|$.

Case 1, $|\hat{\gamma}| < |\gamma|$: $\alpha = \langle\beta,\hat{\gamma}\rangle$ with $|\langle\beta,\hat{\gamma}\rangle| < |\langle\beta,\gamma\rangle|$, contradiction.

Case 2, $|\gamma| < |\hat{\gamma}| < 2|\gamma|$: Let $\delta = \gamma(0)\ldots\gamma(|\hat{\gamma}|-|\gamma|-1)$. Hence, by Lemma 39, $\alpha = \langle\beta,\delta\rangle$ with $|\langle\beta,\delta\rangle| < |\langle\beta,\gamma\rangle|$, contradiction.

Case 3, $2|\gamma| < |\hat{\gamma}|$: Can be reduced to 2. ☐

Lemma 39 *Let α,β,γ be sequences such that $\beta \neq \epsilon$, $|\alpha| \geq |\beta|$, $\alpha\beta = \gamma$, and $\alpha^{\omega} = \gamma^{\omega}$. Then also $\beta^{\omega} = \gamma^{\omega}$.*

Proof: We prove inductively that $\alpha^i\beta^i = \gamma^i$. The claim follows, as $i = |\gamma|$ implies $\beta^i = \beta^{|\gamma|} = \gamma^{|\beta|}$. Base case, $i = 1$: by definition of α,β,γ. Inductive case: assume $\alpha^i\beta^i = \gamma^i$. Therefore, α^i is a prefix, and β^i a suffix of γ^{i+1}. The remaining "gap" has length γ. From $\alpha^{\omega} = \gamma^{\omega}$, we have that α^{i+1} and $(\alpha^{i+2})(0)\ldots(\alpha^{i+2})(\min(|\alpha^{i+2}|,|\gamma^{i+1}|)-1)$ are prefixes of γ^{i+1}. Further, with $\alpha\beta = \gamma$, β is a prefix of α. Hence, we can fill the "gap" with $\alpha\beta$. ☐

Lemma 2 *Let $\alpha = \beta\gamma^{\omega} = \beta'\gamma'^{\omega}$ with $\langle\beta,\gamma\rangle$ minimal for α. Then $|\beta'| \geq |\beta|$ and $|\gamma|$ divides $|\gamma'|$.*

Proof: First, assume $|\beta'| < |\beta|$.

$$\begin{aligned}\beta\gamma^{\omega} = \beta'\gamma'^{\omega} &\Rightarrow \exists 0 \leq i < |\gamma| \,.\, \gamma_{rot} = \gamma[i,|\gamma|-1]\circ\gamma[0,i-1] \wedge {\gamma_{rot}}^{\omega} = \gamma'^{\omega}\\ &\Rightarrow \alpha = \beta'{\gamma_{rot}}^{\omega} \text{ with } |\langle\beta',\gamma_{rot}\rangle| < |\langle\beta,\gamma\rangle|\\ &\Rightarrow \text{contradiction, } \langle\beta,\gamma\rangle \text{ is minimal for } \alpha\end{aligned}$$

Now, assume $|\gamma|$ does not divide $|\gamma'|$. From $\beta\gamma^\omega = \beta'\gamma'^\omega$ we have

$$\exists 0 \leq i < |\gamma'| \,.\, \gamma'_{rot} = \gamma'[i, |\gamma'| - 1] \circ \gamma'[0, i-1] \wedge {\gamma'_{rot}}^\omega = \gamma^\omega$$

Assume further that $|\gamma| < |\gamma'_{rot}|$. Hence, there exist $j > 0$ and γ'' with $0 < |\gamma''| < |\gamma|$ such that $\gamma^j\gamma'' = \gamma'_{rot}$. By Lemma 39, $\gamma''^\omega = {\gamma'_{rot}}^\omega = \gamma^\omega$, and, therefore, $\alpha = \beta\gamma''^\omega$ with $|\langle\beta, \gamma''\rangle| < |\langle\beta, \gamma\rangle|$, a contradiction. The case $|\gamma| > |\gamma'_{rot}|$ is similar. □

B

Raw Data

		NuSMV						Cadence SMV					
		L2S			Live			L2S			Live		
model	property	time	mem	len	time	mem	len	time	mem	len	time	mem	len
1394-3-2	0	1.1	840	–	0.1	82	–	0.3	84	–	er	er	er
	¬ 0	6.8	1223	11	69.5	119	16	7.0	917	11	er	er	er
	1	10.6	1536	–	5.2	496	–	9.6	1032	–	er	er	er
	¬ 1	6.7	1454	11	16.1	157	12	5.7	801	11	er	er	er
1394-4-2	0	123.3	24872	–	1.6	1173	–	0.6	174	–	er	er	er
	¬ 0	396.7	32988	16	to	to	to	212.0	17568	16	er	er	er
	1	680.1	44539	–	548.4	7037	–	306.5	20905	–	er	er	er
	¬ 1	405.6	38455	16	785.0	2356	20	182.6	15159	16	er	er	er
abp4	0	97.1	3471	–	1.0	256	–	107.6	10076	–	1.6	160	–
	L	14.3	841	19	1.0	234	37	21.1	2507	19	0.9	147	34
	¬ L	0.2	149	–	0.1	10	–	0.2	69	–	0.1	16	–
bc57-sensors	¬ 0	205.7	3694	103	139.7	213	112	127.9	8567	103	154.4	1367	109
	0	to	to	to	49.8	180	–	mo	mo	mo	91.2	1421	–
brp	¬ L	0.4	149	1	4.6	46	6	0.6	74	1	4.0	289	2
	¬ L, nv	98.8	1575	24	14.2	122	68	185.9	18421	24	7.0	484	39
	L	to	to	to	0.9	494	–	mo	mo	mo	3.4	214	–
dme5	L	352.8	1432	103	1362.1	356	343	312.6	6834	103	71.7	2020	295
	¬ L	0.9	143	1	10.6	112	1	0.5	117	1	12.8	832	1
	¬ L, nv	315.3	1245	99	1434.8	330	344	313.9	7324	99	50.2	1339	151
dme6	L	956.1	2969	123	to	to	to	1096.7	15143	123	201.9	3650	335
	¬ L	1.4	330	1	27.4	183	1	0.5	103	1	33.8	1465	1
	¬ L, nv	842.6	2564	119	to	to	to	939.5	14534	119	155.4	2403	171
pci	L	mo	mo	mo	140.5	235	23	mo	mo	mo	76.2	1014	25
	¬ L	0.4	224	1	to	to	to	0.3	126	1	1811.6	2534	1
	F L	mo	mo	mo	143.8	139	22	mo	mo	mo	44.8	974	27
prod-cons	0	6.2	218	21	329.9	70	36	4.4	660	21	526.8	1667	53
	¬ 0	16.3	488	26	3.0	311	69	11.5	1406	26	2.6	202	48
	1	1004.1	11862	–	0.2	97	–	744.2	44461	–	0.2	39	–
	¬ 1, nv	1.3	281	21	1.9	250	33	1.5	280	21	0.9	124	65
	2	3.4	220	24	68.0	216	58	4.5	560	24	142.6	567	57
	3	2.8	215	24	7.9	241	42	3.8	614	24	47.7	698	64
	4	1168.2	16054	–	1.2	404	–	958.6	62493	–	6.2	141	–
production-cell	0	8.0	313	–	1.3	222	–	5.5	512	–	3.3	90	–
	¬ 0	4.9	163	83	0.7	300	85	3.4	390	83	0.8	87	84
	1	13.8	381	–	4.1	184	–	7.8	823	–	9.3	94	–
	¬ 1	10.7	314	126	1.4	241	146	6.4	655	126	1.1	87	145
	2	13.9	373	–	3.1	179	–	8.0	827	–	7.0	100	–
	¬ 2	10.4	311	125	1.2	233	126	6.0	665	125	1.0	94	126
srg5	L	0.1	47	–	0.1	9	–	0.1	25	–	0.1	6	–
	¬ L	0.1	30	1	0.4	120	16	0.1	24	1	0.6	20	1
	¬ L, nv	0.2	145	6	0.2	31	15	0.4	120	6	0.2	11	15

Table B.1: "L2S, not tight, ic (none)" versus "Live, not tight" — CPU time [seconds], memory usage [1000 BDD nodes], counterexample length [states]

		L2S		BMC			
				zChaff		MiniSat	
model	property	time	mem	time	mem	time	mem
1394-3-2	¬ 0	7.6	36.7	9.9	30.1	9.4	23.9
	¬ 1	6.7	38.3	8.7	35.3	8.7	25.9
1394-4-2	¬ 0	412.1	748.8	496.0	191.5	361.5	99.6
	¬ 1	406.6	793.8	496.1	186.9	441.9	108.2
abp4	L	7.3	23.7	31.0	30.6	6.2	21.6
bc57-sensors	¬ 0	210.9	159.7	1307.0	353.2	1852.3	256.5
brp	¬ L	0.3	13.6	0.1	1.0	0.1	1.0
	¬ L, nv	102.0	113.0	714.1	172.5	to	to
dme5	L	349.1	71.1	to	to	to	to
	¬ L	1.0	17.4	0.1	1.0	0.1	1.0
	¬ L, nv	360.8	63.3	to	to	to	to
dme6	L	984.4	133.6	to	to	to	to
	¬ L	1.6	18.4	0.1	12.9	0.1	12.4
	¬ L, nv	868.2	115.6	to	to	to	to
pci	L	mo	mo	1453.4	197.8	1642.7	362.7
	¬ L	0.8	17.5	0.3	14.7	0.2	13.4
	F L	mo	mo	562.9	123.7	1061.5	257.6
prod-cons	0	103.2	103.4	19.0	33.0	22.2	27.6
	¬ 0	250.4	175.9	53.0	46.5	534.9	158.2
	¬ 1, nv	61.1	69.7	20.2	33.8	33.7	31.4
	2	61.6	73.9	2.1	24.9	27.7	28.7
	3	62.8	77.4	15.7	31.7	28.7	31.4
production-cell	¬ 0	15.4	25.2	to	to	1325.6	159.6
	¬ 1	to	to	79.5	148.7	281.9	108.2
	¬ 2	328.2	101.1	321.1	172.1	96.8	84.2
srg5	¬ L	0.1	11.3	0.1	1.0	0.1	1.0
	¬ L, nv	1.2	20.1	0.1	1.0	0.1	13.0

Table B.2: "L2S, tight, ic(none), NuSMV" (no model specific order) versus "BMC, tight, NuSMV" — CPU time [seconds], memory usage [megabytes]

model	property	tight time	tight mem	tight len	maxunroll= 2 time	maxunroll= 2 mem	maxunroll= 2 len	maxunroll= 1 time	maxunroll= 1 mem	maxunroll= 1 len	not tight time	not tight mem	not tight len
1394-3-2	0	1.1	840	–	na	na	na	na	na	na	1.1	840	–
	¬ 0	7.6	1185	11	na	na	na	na	na	na	6.8	1223	11
1394-4-2	0	125.5	24875	–	na	na	na	na	na	na	123.3	24872	–
	¬ 0	409.6	34577	16	na	na	na	na	na	na	396.7	32988	16
abp4	L	7.3	480	16	na	na	na	7.3	478	16	14.3	841	19
	¬ L	0.1	120	–	na	na	na	0.1	113	–	0.2	149	–
bc57-sensors	¬ 0	195.9	3876	103	na	na	na	185.5	4172	103	205.7	3694	103
	0	to	to	to	na	na	na	to	to	to	to	to	to
brp	¬ L	0.3	193	1	na	na	na	0.3	171	1	0.4	149	1
	¬ L, nv	102.2	1477	24	na	na	na	103.4	1466	24	98.8	1575	24
	L	to	to	to	na	na	na	to	to	to	to	to	to
dme5	L	348.7	1513	103	na	na	na	392.1	1618	103	352.8	1432	103
	¬ L	1.2	189	1	na	na	na	1.0	174	1	0.9	143	1
	¬ L, nv	360.4	1496	99	na	na	na	348.0	1482	99	315.3	1245	99
dme6	L	983.8	3002	123	na	na	na	957.9	2237	123	956.1	2969	123
	¬ L	1.5	371	1	na	na	na	1.6	348	1	1.4	330	1
	¬ L, nv	866.2	2622	119	na	na	na	864.9	2660	119	842.6	2564	119
pci	L	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo
	¬ L	0.8	409	1	0.7	338	1	0.5	271	1	0.4	224	1
	F L	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo
prod-cons	0	7.1	247	21	na	na	na	na	na	na	6.2	218	21
	¬ 0	18.3	701	26	na	na	na	na	na	na	16.3	488	26
	1	1605.8	15302	–	1615.6	15732	–	1377.7	15332	–	1004.1	11862	–
	¬ 1, nv	2.0	138	21	1.6	561	21	1.5	424	21	1.3	281	21
production-cell	0	23.4	430	–	15.3	325	–	14.4	327	–	8.0	313	–
	¬ 0	10.0	215	81	6.1	235	81	5.7	189	81	4.9	163	83
	¬ 1	to	to	to	9.8	284	81	7.3	263	81	10.7	314	126
	1	to	to	to	33.5	612	–	24.9	529	–	13.8	381	–
	2	859.9	3405	–	25.7	525	–	21.6	470	–	13.9	373	–
	¬ 2	285.2	1730	81	8.2	256	81	6.7	260	81	10.4	311	125
srg5	L	0.2	139	–	0.3	126	–	0.2	89	–	0.1	47	–
	¬ L	0.1	72	1	0.1	55	1	0.1	41	1	0.1	30	1
	¬ L, nv	1.2	191	6	0.8	464	6	0.4	267	6	0.2	145	6

Table B.3: "Tight" versus "maxunroll2" versus "maxunroll1" versus "not tight" ("L2S, ic(none), NuSMV") — CPU time [seconds], memory usage [1000 BDD nodes], counterexample length [states]

model	property	tight time	tight mem	tight len	maxunroll= 2 time	maxunroll= 2 mem	maxunroll= 2 len	maxunroll= 1 time	maxunroll= 1 mem	maxunroll= 1 len	not tight time	not tight mem	not tight len
1394-3-2	0	0.3	84	–	na	na	na	na	na	na	0.3	84	–
	¬ 0	7.1	938	11	na	na	na	na	na	na	7.0	917	11
1394-4-2	0	0.7	177	–	na	na	na	na	na	na	0.6	174	–
	¬ 0	223.6	19628	16	na	na	na	na	na	na	212.0	17568	16
abp4	L	11.2	1260	16	na	na	na	11.1	1254	16	21.1	2507	19
	¬ L	0.2	76	–	na	na	na	0.2	73	–	0.2	69	–
bc57-sensors	0	mo	mo	mo	na	na	na	mo	mo	mo	mo	mo	mo
	¬ 0	162.1	9733	103	na	na	na	149.3	11125	103	127.9	8567	103
brp	L	mo	mo	mo	na	na	na	mo	mo	mo	mo	mo	mo
	¬ L	0.3	86	1	na	na	na	0.3	99	1	0.6	74	1
	¬ L, nv	264.7	23140	24	na	na	na	233.2	20259	24	185.9	18421	24
dme5	L	306.8	7346	103	na	na	na	306.7	7344	103	312.6	6834	103
	¬ L	0.6	131	1	na	na	na	0.5	124	1	0.5	117	1
	¬ L, nv	293.0	5962	99	na	na	na	292.4	5958	99	313.9	7324	99
dme6	L	1115.2	14748	123	na	na	na	1115.4	14745	123	1096.7	15143	123
	¬ L	0.5	107	1	na	na	na	0.5	122	1	0.5	103	1
	¬ L, nv	993.3	13870	119	na	na	na	992.4	14051	119	939.5	14534	119
pci	L	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo
	¬ L	0.5	119	1	0.7	116	1	0.4	123	1	0.3	126	1
	F L	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo	mo
prod-cons	0	10.9	1097	21	na	na	na	na	na	na	4.4	660	21
	¬ 0	28.2	3118	26	na	na	na	na	na	na	11.5	1406	26
	1	994.2	58351	–	992.8	58341	–	1013.5	58795	–	744.2	44461	–
	¬ 1, nv	2.1	409	21	1.8	370	21	1.7	371	21	1.5	280	21
production-cell	0	22.2	1022	–	11.3	801	–	11.3	748	–	5.5	512	–
	¬ 0	11.9	644	81	5.8	500	81	5.9	499	81	3.4	390	83
	1	mo	mo	mo	21.2	1551	–	14.6	1272	–	7.8	823	–
	¬ 1	mo	mo	mo	7.5	811	81	4.8	598	81	6.4	655	126
	¬ 2	1619.0	7056	81	6.5	579	81	4.7	499	81	6.0	665	125
	2	to	to	to	17.1	1164	–	13.4	1054	–	8.0	827	–
srg5	L	0.3	89	–	0.2	47	–	0.2	52	–	0.1	25	–
	¬ L	0.2	54	1	0.2	47	1	0.2	40	1	0.1	24	1
	¬ L, nv	2.5	412	6	1.0	251	6	0.7	242	6	0.4	120	6

Table B.4: "Tight" versus "maxunroll2" versus "maxunroll1" versus "not tight" ("L2S, ic(none), Cadence SMV") — CPU time [seconds], memory usage [1000 BDD nodes], counterexample length [states]

			absref		coi		ic		none	
model	property	tight	time	mem	time	mem	time	mem	time	mem
1394-3-2	0	not tight	0.2	164	1.0	773	1.1	840	1.5	1048
	0	tight	0.2	166	1.1	773	1.1	840	1.5	1046
	$\neg$ 0	not tight	15.2	1131	6.6	1131	6.8	1223	8.6	1557
	$\neg$ 0	tight	14.9	1103	6.7	1103	7.6	1185	8.9	1514
	1	not tight	11.5	1493	10.3	1493	10.6	1536	12.4	1927
	$\neg$ 1	not tight	4.6	718	6.5	1382	6.7	1454	8.1	1773
1394-4-2	0	not tight	5.0	3271	116.5	23128	123.3	24872	151.7	29944
	0	tight	5.0	3270	116.3	23129	125.5	24875	150.2	29943
	$\neg$ 0	not tight	839.4	31266	380.6	30499	396.7	32988	506.1	41353
	$\neg$ 0	tight	847.1	32093	365.5	31024	409.6	34577	526.7	41506
	1	not tight	734.8	42553	662.1	42553	680.1	44539	937.9	53221
	$\neg$ 1	not tight	598.7	37220	386.3	37220	405.6	38455	499.7	50555
abp4	0	not tight	5.9	478	4.9	271	97.1	3471	307.8	7803
	L	not tight	4.5	537	3.0	186	14.3	841	46.4	1820
	L	tight	3.0	548	1.9	150	7.3	480	21.1	1137
	$\neg$ L	not tight	0.1	39	0.1	66	0.2	149	0.4	286
	$\neg$ L	tight	0.1	44	0.1	64	0.1	120	0.2	172
bc57-sensors	0	not tight	251.2	1254	to	to	na	na	to	to
	0	tight	338.8	1438	to	to	na	na	to	to
	$\neg$ 0	not tight	154.9	2854	186.2	3595	na	na	205.7	3694
	$\neg$ 0	tight	158.1	3109	181.8	4098	na	na	195.9	3876
brp	L	not tight	2.9	287	to	to	to	to	to	to
	L	tight	7.7	254	to	to	to	to	to	to
	$\neg$ L	not tight	0.1	67	0.2	119	0.4	149	0.3	162
	$\neg$ L	tight	0.3	95	0.3	157	0.3	193	0.5	206
	$\neg$ L, nv	not tight	77.2	672	23.5	672	98.8	1575	210.4	3104
	$\neg$ L, nv	tight	97.2	700	34.7	700	102.2	1477	294.4	4313
dme5	L	not tight	666.6	1012	na	na	352.8	1432	1510.5	4997
	L	tight	698.7	797	na	na	348.7	1513	1586.0	5025
	$\neg$ L	not tight	6.0	418	na	na	0.9	143	1.1	176
	$\neg$ L	tight	8.2	400	na	na	1.2	189	1.1	210
	$\neg$ L, nv	not tight	625.2	723	na	na	315.3	1245	1392.5	4054
	$\neg$ L, nv	tight	648.7	871	na	na	360.4	1496	1466.8	3859
dme6	L	not tight	1260.1	1053	na	na	956.1	2969	to	to
	L	tight	1398.2	1256	na	na	983.8	3002	to	to
	$\neg$ L	not tight	8.4	337	na	na	1.4	330	1.4	337
	$\neg$ L	tight	8.2	340	na	na	1.5	371	1.6	380
	$\neg$ L, nv	not tight	1165.0	1177	na	na	842.6	2564	to	to
	$\neg$ L, nv	tight	1313.3	1301	na	na	866.2	2622	to	to
pci	L	not tight	mo	mo	na	na	mo	mo	mo	mo
	L	tight	mo	mo	na	na	mo	mo	mo	mo
	$\neg$ L	not tight	1.7	224	na	na	0.4	224	0.5	236
	$\neg$ L	tight	3.5	409	na	na	0.8	409	0.9	420
	$\mathbf{F}$ L	not tight	mo	mo	na	na	mo	mo	mo	mo
	$\mathbf{F}$ L	tight	mo	mo	na	na	mo	mo	mo	mo
prod-cons	0	not tight	6.6	218	na	na	6.2	218	7.4	341
	0	tight	7.5	247	na	na	7.1	247	11.4	379
	$\neg$ 0	not tight	11.6	430	na	na	16.3	488	23.0	859
	$\neg$ 0	tight	17.9	583	na	na	18.3	701	32.1	1093
	1	not tight	0.3	215	477.4	6125	1004.1	11862	2278.6	23389
	1	tight	0.8	398	682.5	8153	1605.8	15302	2315.6	23408
	$\neg$ 1, nv	not tight	1.1	378	0.5	378	1.3	281	1.7	173
	$\neg$ 1, nv	tight	2.0	558	0.8	558	2.0	138	1.9	146
	2	not tight	4.2	558	na	na	3.4	220	4.7	332
	3	not tight	2.4	158	na	na	2.8	215	3.5	311
	4	not tight	3.9	196	482.0	8215	1168.2	16054	2286.1	29144
production-cell	0	not tight	7.3	249	na	na	na	na	8.0	313
	0	tight	21.9	350	na	na	na	na	23.4	430
	$\neg$ 0	not tight	4.9	249	na	na	na	na	4.9	163
	$\neg$ 0	tight	11.1	523	na	na	na	na	10.0	215
	1	not tight	12.8	321	na	na	na	na	13.8	381
	1	tight	to	to	na	na	na	na	to	to
	$\neg$ 1	not tight	11.8	264	na	na	na	na	10.7	314
	$\neg$ 1	tight	mo	mo	na	na	na	na	to	to
	2	not tight	12.1	360	na	na	na	na	13.9	373
	2	tight	763.1	3374	na	na	na	na	859.9	3405
	$\neg$ 2	not tight	11.4	304	na	na	na	na	10.4	311
	$\neg$ 2	tight	to	to	na	na	na	na	285.2	1730
srg5	L	not tight	0.2	47	na	na	0.1	47	0.1	50
	L	tight	0.3	139	na	na	0.2	139	0.2	129
	$\neg$ L	not tight	0.2	30	na	na	0.1	30	0.1	31
	$\neg$ L	tight	0.2	72	na	na	0.1	72	0.1	75
	$\neg$ L, nv	not tight	0.6	145	na	na	0.2	145	0.2	160
	$\neg$ L, nv	tight	7.1	673	na	na	1.2	191	1.4	482

Table B.5: Degrees of variable optimization (“L2S, NuSMV”) — CPU time [seconds], memory usage [1000 BDD nodes]

			absref		coi		ic		none	
model	property	tight	time	mem	time	mem	time	mem	time	mem
1394-3-2	0	not tight	0.3	79	0.3	89	0.3	84	0.4	81
	0	tight	0.3	79	0.3	89	0.3	84	0.7	110
	¬ 0	not tight	8.1	868	6.5	879	7.0	917	9.4	1201
	¬ 0	tight	8.3	887	6.7	899	7.1	938	9.8	1090
	1	not tight	9.3	949	8.2	949	9.6	1032	11.8	1345
	¬ 1	not tight	4.8	538	5.4	805	5.7	801	7.2	952
1394-4-2	0	not tight	0.5	153	0.6	173	0.6	174	0.8	157
	0	tight	0.5	143	0.8	132	0.7	177	0.7	167
	¬ 0	not tight	205.2	18168	202.7	18374	212.0	17568	294.3	25739
	¬ 0	tight	232.8	18473	216.6	18703	223.6	19628	305.8	25897
	1	not tight	299.9	20012	296.7	20012	306.5	20905	445.4	28847
	¬ 1	not tight	134.2	9285	178.1	14799	182.6	15159	269.4	22222
abp4	0	not tight	9.5	791	7.6	791	107.6	10076	313.3	27628
	L	not tight	7.2	600	4.7	600	21.1	2507	57.1	5477
	L	tight	4.8	337	2.9	337	11.2	1260	34.0	3337
	¬ L	not tight	0.1	31	0.1	42	0.2	69	0.5	178
	¬ L	tight	0.1	46	0.1	38	0.2	76	0.4	150
bc57-sensors	0	not tight	91.1	2779	mo	mo	na	na	mo	mo
	0	tight	122.3	3529	mo	mo	na	na	mo	mo
	¬ 0	not tight	94.4	5959	126.7	8486	na	na	127.9	8567
	¬ 0	tight	127.8	7950	151.5	11001	na	na	162.1	9733
brp	L	not tight	21.7	991	mo	mo	mo	mo	mo	mo
	L	tight	28.0	1225	mo	mo	mo	mo	mo	mo
	¬ L	not tight	0.1	49	0.2	58	0.6	74	0.3	71
	¬ L	tight	0.2	50	0.3	82	0.3	86	0.3	90
	¬ L, nv	not tight	185.3	6534	59.9	6534	185.9	18421	452.9	39010
	¬ L, nv	tight	296.1	9611	94.1	9611	264.7	23140	592.6	51752
dme5	L	not tight	127.1	837	na	na	312.6	6834	1295.8	23036
	L	tight	135.8	1059	na	na	306.8	7346	1312.5	22279
	¬ L	not tight	0.2	59	na	na	0.5	117	0.8	132
	¬ L	tight	0.2	64	na	na	0.6	131	0.5	136
	¬ L, nv	not tight	197.5	962	na	na	313.9	7324	1053.7	23338
	¬ L, nv	tight	206.6	1062	na	na	293.0	5962	958.9	23542
dme6	L	not tight	345.3	1407	na	na	1096.7	15143	to	to
	L	tight	344.6	1681	na	na	1115.2	14748	to	to
	¬ L	not tight	0.2	96	na	na	0.5	103	0.5	125
	¬ L	tight	0.3	113	na	na	0.5	107	0.6	137
	¬ L, nv	not tight	512.0	1428	na	na	939.5	14534	to	to
	¬ L, nv	tight	519.0	1640	na	na	993.3	13870	to	to
pci	L	not tight	mo	mo	na	na	mo	mo	mo	mo
	L	tight	mo	mo	na	na	mo	mo	mo	mo
	¬ L	not tight	0.2	69	na	na	0.3	126	0.4	96
	¬ L	tight	0.3	129	na	na	0.5	119	0.6	147
	F L	not tight	mo	mo	na	na	mo	mo	mo	mo
	F L	tight	mo	mo	na	na	mo	mo	mo	mo
prod-cons	0	not tight	5.0	660	na	na	4.4	660	7.2	1077
	0	tight	11.6	1097	na	na	10.9	1097	16.5	2226
	¬ 0	not tight	22.2	1406	na	na	11.5	1406	20.7	2899
	¬ 0	tight	53.1	3118	na	na	28.2	3118	45.2	4275
	1	not tight	0.5	85	383.4	22425	744.2	44461	mo	mo
	1	tight	1.4	175	590.7	30647	994.2	58351	mo	mo
	¬ 1, nv	not tight	1.7	139	0.7	139	1.5	280	2.4	409
	¬ 1, nv	tight	2.5	210	1.0	210	2.1	409	2.5	473
	2	not tight	10.9	684	na	na	4.5	560	7.2	924
	3	not tight	7.4	614	na	na	3.8	614	6.1	819
	4	not tight	5.6	502	466.9	30424	958.6	62493	mo	mo
production-cell	0	not tight	5.4	444	na	na	na	na	5.5	512
	0	tight	19.2	1196	na	na	na	na	22.2	1022
	¬ 0	not tight	4.2	401	na	na	na	na	3.4	390
	¬ 0	tight	12.3	753	na	na	na	na	11.9	644
	1	not tight	7.5	760	na	na	na	na	7.8	823
	1	tight	mo	mo	na	na	na	na	mo	mo
	¬ 1	not tight	7.7	661	na	na	na	na	6.4	655
	¬ 1	tight	mo	mo	na	na	na	na	mo	mo
	2	not tight	7.1	723	na	na	na	na	8.0	827
	2	tight	to	to	na	na	na	na	to	to
	¬ 2	not tight	7.6	671	na	na	na	na	6.0	665
	¬ 2	tight	to	to	na	na	na	na	1619.0	7056
srg5	L	not tight	0.2	25	na	na	0.1	25	0.1	24
	L	tight	0.5	89	na	na	0.3	89	0.4	68
	¬ L	not tight	0.2	24	na	na	0.1	24	0.1	23
	¬ L	tight	0.4	54	na	na	0.2	54	0.2	55
	¬ L, nv	not tight	1.3	120	na	na	0.4	120	0.4	119
	¬ L, nv	tight	10.0	1787	na	na	2.5	412	2.5	490

Table B.6: Degrees of variable optimization ("L2S, Cadence SMV") — CPU time [seconds], memory usage [1000 BDD nodes]

		NuSMV						Cadence SMV					
		tight			not tight			tight			not tight		
model	property	time	mem	len	time	mem	len	time	mem	len	time	mem	len
1394-3-2	0	0.1	92	–	0.1	86	–	er	er	er	er	er	er
	¬ 0	105.5	199	17	70.9	180	16	er	er	er	er	er	er
1394-4-2	0	1.6	1176	–	1.6	1174	–	er	er	er	er	er	er
	¬ 0	to	to	to	to	to	to	er	er	er	er	er	er
abp4	L	2.0	45	43	0.9	216	37	2.4	261	36	0.9	147	34
	¬ L	0.1	26	–	0.1	11	–	0.2	30	–	0.1	16	–
bc57-sensors	0	139.8	154	–	43.7	159	–	169.3	1874	–	91.2	1421	–
	¬ 0	419.7	469	112	108.8	154	104	319.2	2030	106	154.4	1367	109
brp	L	4.4	130	–	0.9	467	–	4.3	287	–	3.4	214	–
	¬ L	19.2	269	10	4.6	204	6	5.1	274	4	4.0	289	2
	¬ L, nv	51.7	94	70	13.5	347	56	9.3	504	50	7.0	484	39
dme5	L	1586.9	390	273	1229.2	447	343	224.6	2462	125	71.7	2020	295
	¬ L	446.6	240	174	10.8	218	1	150.1	1982	151	12.8	832	1
	¬ L, nv	2161.9	580	100	1129.0	256	344	268.3	2047	121	50.2	1339	151
dme6	¬ L	1458.5	363	194	23.4	310	1	503.1	3683	171	33.8	1465	1
	¬ L, nv	to	to	to	to	to	to	875.4	3947	141	155.4	2403	171
	L	to	to	to	to	to	to	682.9	4708	145	201.9	3650	335
pci	¬ L	to	to	to	to	to	to	mo	mo	mo	1811.6	2534	1
	L	to	to	to	379.4	323	23	to	to	to	76.2	1014	25
	F L	to	to	to	443.7	142	22	mo	mo	mo	44.8	974	27
prod-cons	0	443.7	161	38	359.0	406	36	896.2	2545	43	526.8	1667	53
	¬ 0	16.6	163	58	2.8	46	40	33.5	648	63	2.6	202	48
	1	1.3	89	–	0.1	81	–	3.0	134	–	0.2	39	–
	¬ 1, nv	6.3	261	34	1.6	18	33	7.9	197	70	0.9	124	65
production-cell	¬ 0	to	to	to	0.9	298	85	1817.7	5921	84	0.8	87	84
	0	to	to	to	1.4	221	–	1544.9	3819	–	3.3	90	–
	¬ 1	to	to	to	1.6	211	146	mo	mo	mo	1.1	87	145
	1	to	to	to	3.9	249	–	mo	mo	mo	9.3	94	–
	¬ 2	to	to	to	1.3	340	126	mo	mo	mo	1.0	94	126
	2	to	to	to	2.9	270	–	mo	mo	mo	7.0	100	–
srg5	L	0.6	186	–	0.1	8	–	4.5	88	–	0.1	6	–
	¬ L	12.9	65	27	0.4	78	11	63.6	394	5	0.6	20	1
	¬ L, nv	9.7	62	22	0.1	22	11	38.0	676	19	0.2	11	15

Table B.7: “Tight” versus “not tight” (“Live”) — CPU time [seconds], memory usage [1000 BDD nodes], counterexample length [states]

List of Figures

2.1 The semantics of PLTLB . . . 16
2.2 A general scheme for abstraction refinement . . . 25
2.3 No tightness with forced fairness in bit-set degeneralization . . . 25

3.1 A generic lasso-shaped counterexample . . . 27
3.2 A 2-bit counter with self-loops . . . 29
3.3 Translating simple liveness: NuSMV code of 2-bit counter with self-loops . . . 29
3.4 A run of the state-recording translation for the generic counterexample . . . 31
3.5 Translating fairness: NuSMV code of 2-bit counter with self-loops . . . 33
3.6 Translating hierarchy: NuSMV code of mutex with Büchi specification . . . 37
3.7 Büchi automaton for $\neg\mathbf{G}((s = try) \rightarrow (\mathbf{F}(s = crit)))$ . . . 38

4.1 Example: token passing [BJNT00] . . . 50
4.2 The reduction preserves boundedness of local depth. . . . 51
4.3 No shortest counterexamples for pushdown systems . . . 58

5.1 Scenarios with shortest and non-optimal counterexample . . . 68
5.2 Model M and Büchi automaton $B^{p \wedge \mathbf{XG}q}_{GPVW}$ resulting in non-optimal counterexample 71
5.3 Counterexamples with excess length linear in operator depth of formula . . . 73

6.1 Abstraction refinement for loop detection . . . 91
6.2 State-recording translation can succeed with no variables in loop detection . . . 92
6.3 Removing output variables from loop detection can shorten counterexamples . 94

7.1 Forward jumping counter . . . 97
7.2 Charts: forward jumping counter . . . 98
7.3 Charts: L2S versus Live . . . 103
7.4 Charts: L2S versus BMC . . . 104
7.5 Charts: comparing degrees of tightness . . . 106
7.6 Charts: comparing degrees of variable optimization (CPU time) . . . 107
7.7 Charts: comparing degrees of variable optimization (memory usage) . . . 107

7.8 Charts: tight versus not tight (Live) 109

List of Tables

2.1 Definition of subformulae 16
2.2 Property-dependent part of a Büchi automaton constructed with KPR [KPR98] 19

3.1 Deriving a bound on the number of transitions in the transitive closure 39
3.2 BDD sizes for Eqn. (3.2) ($*$ = memory limit of 512 MB reached). 40

4.1 Time complexity for alg. 3 in [EHRS00a] when applied to pushdown system . . 57

5.1 Property-dependent part of a tight Büchi automaton 77

7.1 Real-world examples: models 99
7.2 Real-world examples: templates of the properties 99

B.1 Raw data: L2S versus Live 119
B.2 Raw data: L2S versus BMC 120
B.3 Raw data: tight vs maxunroll2/1 vs not tight (L2S, NuSMV) 121
B.4 Raw data: tight vs maxunroll2/1 vs not tight (L2S, Cadence SMV) 122
B.5 Raw data: degrees of variable optimization (L2S, NuSMV) 123
B.6 Raw data: degrees of variable optimization (L2S, Cadence SMV) 124
B.7 Raw data: tight versus not tight (Live) 125

Bibliography

[ABBL03] L. Aceto, P. Bouyer, A. Burgueño, and K. Larsen. The power of reachability testing for timed automata. *Theor. Comput. Sci.*, 300(1-3):411–475, 2003. 65

[Acc] Accellera. Property specification language reference manual, version 1.1. http://www.eda.org/vfv/docs/PSL-v1.1.pdf. 115

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126:183–235, 1994. 4, 47, 60, 61, 65, 66, 112

[ADS86] B. Alpern, A. Demers, and F. Schneider. Safety without stuttering. *Inf. Process. Lett.*, 23(4):177–180, 1986. 20

[AJN$^+$04] P. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular model checking for LTL(MSO). In Alur and Peled [AP04], pages 348–360. 66

[AJNd03] P. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In Hunt Jr. and Somenzi [HS03], pages 236–248. 47

[AK86] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986. 51, 114

[Alu99] R. Alur. Timed automata. In Halbwachs and Peled [HP99], pages 8–22. 114

[AP04] R. Alur and D. Peled, editors. *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *LNCS*. Springer, 2004. 131, 134

[AS85] B. Alpern and F. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. 19, 20

[AS87] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987. 2, 20

[AS04] M. Awedh and F. Somenzi. Proving more properties with bounded model checking. In Alur and Peled [AP04], pages 96–108. 3, 9, 24, 83, 93

[AS06] M. Awedh and F. Somenzi. Termination criteria for bounded model checking: Extensions and comparison. In A. Biere and O. Strichman, editors, *Proceedings of the Third International Workshop on Bounded Model Checking (BMC 2005), Edinburgh, UK, 11 July 2005*, ENTCS, 144(1), pages 51–66. Elsevier, 2006. 93, 115

[AVARB+01] Y. Abarbanel-Vinov, N. Aizenbud-Reshef, I. Beer, C. Eisner, D. Geist, T. Heyman, I. Reuveni, E. Rippel, I. Shitsevalov, Y. Wolfsthal, and T. Yatzkar-Haham. On the effective deployment of functional formal verification. *Formal Methods in System Design*, 19(1):35–44, 2001. 3, 8

[BAS02] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In R. Cleaveland and H. Garavel, editors, *Formal Methods for Industrial Critical Systems, Proceedings of the 7th International ERCIM Workshop, FMICS'02, Málaga, Spain, July 12–13, 2002*, ENTCS, 66(2). Elsevier, 2002. 5, 40

[BBČ02] J. Barnat, L. Brim, and I. Černá. Property driven distribution of nested DFS. In M. Leuschel and U. Ultes-Nitsche, editors, *Proceeding of the 3rd International Workshop on Verification and Computational Logic (VCL'2002)*, DSSE Technical Report, DSSE-TR-2002-5, pages 1–10. Dept. of Electronics and Computer Science, University of Southampton, UK, 2002. 113

[BBDL98] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In Hu and Vardi [HV98], pages 184–194. 44

[BBF+01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001. 1, 2, 9

[BC03] M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *LNCS*, pages 18–33. Springer, 2003. 8, 17, 24, 45, 74, 75, 82, 84, 112

[BCC+99] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation, Proceedings of the 36th ACM/IEEE Conference, DAC'99, New Orleans, Louisiana, United States, June 21–25, 1999*, pages 317–320. ACM Press, 1999. 9, 23

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, Proceedings of the 5th International Conference, TACAS '99, Amsterdam, The Netherlands, March 22–28, 1999*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999. 2, 3, 4, 9, 23, 24, 28, 44, 45, 74, 87, 92, 93

[BCF01] G. Berry, H. Comon, and A. Finkel, editors. *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *LNCS*. Springer, 2001. 135, 138, 139

[BCLR04] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In E. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004. 3, 8

[BCM+92] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992. 9, 45, 83, 97, 98, 100, 102, 108

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifiying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In Halbwachs and Peled [HP99], pages 60–71. 9, 23, 93, 95

[BCZ99] A. Biere, E. Clarke, and Y. Zhu. Multiple state and single state tableaux for combining local and global model checking. In E.-R. Olderog and B. Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*, pages 163–179. Springer, 1999. 2, 4, 23

[BDEGW03] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at IBM. *Formal Methods in System Design*, 22(2):101–108, 2003. 2, 8

[BDGP98] B. Bérard, V. Diekert, P. Gastin, and A. Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2–3):145–182, 1998. 60, 62, 64, 65

[BDL04] G. Behrmann, A. David, and K. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004. 2

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997. 4, 47, 52, 55, 66, 112

[Ben01] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001* [DAC01], pages 244–248. 2, 8

[BF90] S. Bose and A. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In L. Claesen, editor, *Proceedings of the IFIP WG 10.2/WG 10.5 International Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, November 1989*, pages 151–158. North-Holland, 1990. 9

[BFH+01] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in UPPAAL. In Margaria and Yi [MY01], pages 174–188. 65

[BGG02] S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In Brinksma and Larsen [BL02], pages 65–77. 86, 90, 94, 95, 108, 113

[BHJ^{+}06] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. 2006. Submitted. 8, 113

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In Alur and Peled [AP04], pages 372–386. 65, 114

[Bie97] A. Biere. μcke — efficient μ-calculus model checking. In Grumberg [Gru97], pages 468–471. 8

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla [ES00], pages 403–418. 2, 4, 47, 49, 50, 51, 66, 112, 127

[BKA02] J. Baumgartner, A. Kühlmann, and J. Abraham. Property checking via structural analysis. In Brinksma and Larsen [BL02], pages 151–165. 88, 93, 94, 113

[BL02] E. Brinksma and K. Larsen, editors. *Computer Aided Verification, Proceedings of the 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27–31, 2002*, volume 2404 of *LNCS*. Springer, 2002. 133, 134, 135

[BL03] B. Batson and L. Lamport. High-level specifications: Lessons from industry. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, volume 2852 of *LNCS*, pages 242–261. Springer, 2003. 1

[BLW04a] B. Boigelot, A. Legay, and P. Wolper. Omega-regular model checking. In Jensen and Podelski [JP04], pages 561–575. 4, 47, 51

[BLW04b] A. Bouajjani, A. Legay, and P. Wolper. Handling liveness properties in ($\omega-$)regular model checking. In *INFINITY'04*, 2004. 66

[BR02] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In POPL'02 [POP02], pages 1–3. 2, 8, 10, 94

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. 9, 22, 23, 39

[Bry91] R. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Computers*, 40(2):205–213, 1991. 23

[BSV93] F. Balarin and A. Sangiovanni-Vincentelli. An iterative approach to language containment. In Courcoubetis [Cou93], pages 29–40. 10, 24, 94, 95

[Büc62] J. Büchi. On a decision method in restricted second order arithmetic. In E. Nagel, P. Suppes, and A. Tarski, editors, *Logic, Methodology and Philosophy in Science: Proceedings of the 1960 International Congress*, pages 1–11. Stanford University Press, 1962. 17

[Bur89] J. Burch. Modeling timing assumptions with trace theory. In *Proceedings of the 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD'89, Cambridge, MA, USA, October 2–4, 1989*, pages 208–211. IEEE Computer Society Press, 1989. 43

[Bur91] J. Burch. Verifying liveness properties by verifying safety properties. In Clarke and Kurshan [CK91], pages 224–232. 43

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77), Los Angeles, CA, USA, January 17-19, 1977*, pages 238–252. ACM Press, 1977. 9

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Brinksma and Larsen [BL02], pages 359–364. 4, 8, 10, 28, 94, 100

[CCG+04] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004. 10, 94

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000. 100

[CCO+] R. Cavada, A. Cimatti, E. Olivetti, G. Keighren, M. Pistore, and M. Roveri. NuSMV 2.2 user manual. `http://nusmv.irst.itc.it/NuSMV/userman/v22/nusmv.pdf`. 92, 101

[CDK93] E. Clarke, I. Draghicescu, and R. Kurshan. A unified approch for showing language inclusion and equivalence between various types of omega-automata. *Inf. Process. Lett.*, 46(6):301–308, 1993. 18

[CE82] E. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer, 1982. 8, 38

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. 8, 9

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In Berry et al. [BCF01], pages 436–453. 3

[CGH97] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997. 8, 18, 83, 102

[CGJ+03] E. 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. 3, 10, 90, 94, 95

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. 9, 24, 86, 92, 94, 95

[CGMZ95] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation, 32nd ACM/IEEE Conference, DAC'95, San Francisco, California, USA, June 12-16, 1995, Proceedings*, pages 427–432. ACM Press, 1995. 22, 45, 108, 112

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. 2, 9, 10, 12, 59, 60, 88, 91

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Comput. Syst. Sci.*, 8(2):117–141, 1974. 14

[Cim05] A. Cimatti, 2005. Personal communication. 108

[CK91] E. Clarke and R. Kurshan, editors. *Computer Aided Verification, Proceedings of the 2nd International Workshop, CAV'90, New Brunswick, NJ, USA, June 18–21, 1990*, volume 531 of *LNCS*. Springer, 1991. 135, 136, 145

[CKL04] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In Jensen and Podelski [JP04], pages 168–176. 10

[CKOS05] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005. 24, 31, 93

[Cla06] K. Claessen, 2006. Personal communication. 113

[CMB91] O. Coudert, J. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Clarke and Kurshan [CK91], pages 23–32. 9

[Cou93] C. Courcoubetis, editor. *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *LNCS*. Springer, 1993. 134, 139

[Cou99] J. Couvreur. On-the-fly verification of linear temporal logic. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*, volume 1708 of *LNCS*, pages 253–271. Springer, 1999. 83

[CPR05] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In C. Hankin and I. Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *LNCS*, pages 87–101. Springer, 2005. 65, 66

[CRS04] A. Cimatti, M. Roveri, and D. Sheridan. Bounded verification of past LTL. In Hu and Martin [HM04], pages 245–259. 8, 10, 24, 45, 74, 108

[CS01] E. Clarke and B. Schlingloff. Model checking. In J Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1635–1790. Elsevier and MIT Press, 2001. 9, 10

[CV03] E. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 208–224. Springer, 2003. 3, 45

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992. 3, 18, 28, 44, 45, 113

[DAC01] *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001. 133, 149

[DGV99] M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In Halbwachs and Peled [HP99], pages 249–260. 83, 84

[Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In J. Allen and F. Leighton, editors, *Prodeedings of the Fifth MIT Conference on Advanced Research in VLSI, Cambridge, MA, USA, March, 1988*, pages 50–65. MIT Press, 1988. 43

[EH86] E. Emerson and J. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986. 8

[EH00] K. Etessami and G. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, volume 1877 of *LNCS*, pages 153–167. Springer, 2000. 83

[EHRS00a] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In Emerson and Sistla [ES00], pages 232–247. 4, 47, 52, 55, 56, 57, 58, 112, 129

[EHRS00b] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. Technical Report TUM-I0002, Institut für Informatik, Technische Universität München, 2000. 56

[EJ06] S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In A. Valmari, editor, *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, volume 3925 of *LNCS*, pages 1–18. Springer, 2006. 113

[EL87] E. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987. 2

[Eme83] E. Emerson. Alternative semantics for temporal logics. *Theor. Comput. Sci.*, 26:121–130, 1983. 20

[Eme90] A. Emerson. Temporal and modal logic. In van Leeuwen [vL90], pages 995–1072. 7, 8, 15, 18

[ES] N. Eén and N. Sörensson. MiniSat. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html. 100

[ES00] A. Emerson and P. Sistla, editors. *Computer Aided Verification, Proceedings of the 12th International Conference, CAV 2000, Chicago, IL, USA, July 15–19, 2000*, volume 1855 of *LNCS*. Springer, 2000. 134, 137, 146

[ES01] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In Berry et al. [BCF01], pages 324–336. 47, 55

[ES03] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. In O. Strichman and A. Biere, editors, *Bounded Model Checking, First International Workshop, BMC 2003, Boulder, CO, July 13, 2003, Proceedings*, ENTCS, 89(4), pages 543–560. Elsevier, 2003. 24, 87, 93, 94, 113

[ES04] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004. 100

[Flo67] R. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967. 7

[FMPT01] A. Fuxman, J. Mylopoulos, M. Pistore, and P. Traverso. Model checking early requirements specifications in tropos. In *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*, pages 174–181. IEEE Computer Society, 2001. 8

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems (extended abstract). In F. Moller, editor, *INFINITY'97*, volume 9 of *ENTCS*. Elsevier, 1997. 4, 47, 55, 112

[Gab89] D. Gabbay. The declarative past and imperative future. In *Temporal Logic in Specification, Altrinchamm UK, April 8-10, 1987, Proceedings*, volume 398 of *LNCS*, pages 409–448. Springer, 1989. 74

[GD98] S. Govindaraju and D. Dill. Verification by approximate forward and backward reachability. In H. Yasuura, editor, *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'98), San Jose, CA, USA, November 8–12, 1998*, pages 366–370. ACM, 1998. 94, 95

[Gei01] M. Geilen. On the construction of monitors for temporal logic properties. In Havelund and Roşu [HR01b]. 84

[GGA05] M. Ganai, A. Gupta, and P. Ashar. Beyond safety: customized SAT-based model checking. In W Joyner, G. Martin, and A. Kahng, editors, *Proceedings of the 42nd Design Automation Conference, DAC 2005, San Diego, CA, USA, June 13-17, 2005*, pages 738–743. ACM, 2005. 2

[GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering, Proceedings of the 16th IEEE International Conference, ASE 2001, 26–29 November*

2001, Coronado Island, San Diego, CA, USA, pages 412–416. IEEE Computer Society, 2001. 84

[GK05] A. Groce and D. Kröning. Making the most of BMC counterexamples. In A. Biere and O. Strichman, editors, *Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC 2004), MA, July 18, 2004*, ENTCS, 119(2), pages 71–84. Elsevier, 2005. 45, 115

[GMZ04] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In S. Graf and L. Mounier, editors, *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *LNCS*, pages 92–108. Springer, 2004. 45, 83

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In Berry et al. [BCF01], pages 53–65. 84, 114

[GO03] P. Gastin and D. Oddoux. LTL with past and two-way very-weak alternating automata. In B. Rovan and P. Vojtás, editors, *Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003, Proceedings*, volume 2747 of *LNCS*, pages 439–448. Springer, 2003. 8, 84

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In Courcoubetis [Cou93], pages 438–449. 10

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal basis of fairness. In POPL'80 [POP80], pages 163–173. 8

[GPVW96] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1996. 4, 38, 71, 72, 73, 82, 83, 84, 112, 114, 115

[Gro05] A. Groce. *Error Explanation and Fault Localization with Distance Metrics*. PhD thesis, Carnegie Mellon University, 2005. 45

[Gru97] O. Grumberg, editor. *Computer Aided Verification, Proceedings of the 9th International Conference, CAV'97, Haifa, Israel, June 22–25, 1997*, volume 1254 of *LNCS*. Springer, 1997. 134, 139

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In Grumberg [Gru97], pages 72–83. 10, 94

[Hel06] K. Heljanko, 2006. Personal communication. 113

[HJ00] W. Hunt Jr. and S. Johnson, editors. *Formal Methods in Computer-Aided Design, Proceedings of the Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1–3, 2000*, volume 1954 of *LNCS*. Springer, 2000. 146, 148

[HJL] K. Heljanko, T. Junttila, and T. Latvala. CAV submission source code. http://www.tcs.hut.fi/~tjunttil/experiments/CAV05/. 100

[HJL05] K. Heljanko, T. Junttila, and T. Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *LNCS*, pages 98–111. Springer, 2005. 9, 10, 24, 79, 82, 93, 100, 108, 112, 115

[HJMS02] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In POPL'02 [POP02], pages 58–70. 10, 94

[HK05] H. Hansen and A. Kervinen. Minimal counterexamples in linear memory and polynomial time. Manuscript, 2005. 45

[HKQ98] T. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. In Hu and Vardi [HV98], pages 195–206. 2, 4, 23

[HLR94] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21–25 June, 1993*, Workshops in Computing, pages 83–96. Springer, 1994. 84

[HM04] A. Hu and A. Martin, editors. *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, TX, USA, November 14–17, 2004, Proceedings*, volume 3312 of *LNCS*. Springer, 2004. 136, 143

[Hoa69] C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. 7

[Hol88] G. Holzmann. An improved protocol reachability analysis technique. *Softw., Pract. Exper.*, 18(2):137–161, 1988. 96

[Hol98] G. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998. 96

[Hol03] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. 8, 9, 10, 14, 20, 96

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI Series*, pages 477–498. Springer, 1985. 7

[HP99] N. Halbwachs and D. Peled, editors. *Computer Aided Verification, Proceedings of the 11th International Conference, CAV'99, Trento, Italy, July 6–10, 1999*, volume 1633 of *LNCS*. Springer, 1999. 131, 133, 137, 143

[HR01a] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification, Proceedings of the First International Workshop, RV'01, Paris, France, 23 July, 2001* [HR01b]. 84

[HR01b] K. Havelund and G. Roşu, editors. *Runtime Verification, Proceedings of the First International Workshop, RV'01, Paris, France, 23 July, 2001*, ENTCS, 55(2). Elsevier, 2001. 138, 140

[HR02] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In Katoen and Stevens [KS02], pages 342–356. 2, 84

[HS03] W. Hunt Jr. and F. Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *LNCS*. Springer, 2003. 131, 144

[HV98] A. Hu and M. Vardi, editors. *Computer Aided Verification, Proceedings of the 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28 – July 2, 1998*, volume 1427 of *LNCS*. Springer, 1998. 132, 140, 149

[INH96] H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In Rutenbar and Otten [RO96], pages 82–87. 2, 4, 23, 98

[JJ90] C. Jard and T. Jéron. On-line model checking for finite linear temporal logic specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *LNCS*, pages 189–196. Springer, 1990. 44

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1785 of *LNCS*, pages 220–234. Springer, 2000. 51, 82

[Jon04] N. Jones. Liveness manifesto. In Podelski et al. [PSZ]. 1

[JP04] K. Jensen and A. Podelski, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *LNCS*. Springer, 2004. 134, 136, 146

[JPO95] L. Jagadeesan, C. Puchol, and J. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In P. Wolper, editor, *Computer Aided Verification, Proceedings of the 7th International Conference, Liège, Belgium, July, 3–5, 1995*, volume 939 of *LNCS*, pages 127–140. Springer, 1995. 84

[JR00] D. Jackson and M. Rinard. Software analysis: a roadmap. In *ICSE — Future of SE Track*, pages 133–145, 2000. 8

[JRS02] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In Katoen and Stevens [KS02], pages 445–459. 45, 115

[Kam68] J. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California at Los Angeles, 1968. 8

[Kin94] E. Kindler. Safety and liveness properties: A survey. *Bulletin of the EATCS*, 53:268–272, 1994. 20

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.*, 256(1-2):93–112, 2001. 2, 4, 47, 112

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983. 8, 43

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *LNCS*, pages 1–16. Springer, 1998. 4, 12, 16, 18, 19, 24, 73, 74, 75, 79, 83, 84, 100, 101, 108, 112, 114, 115, 129

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language, Second Edition, ANSI C*. Prentice-Hall, 1988. 36

[Kro99] T. Kropf. *Introduction to Formal Hardware Verification*. Springer, 1999. 9

[KS02] J.-P. Katoen and P. Stevens, editors. *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 8th International Conference, TACAS 2002, Grenoble, France, April 8–12, 2002*, volume 2280 of *LNCS*. Springer, 2002. 141

[KS03] D. Kröning and O. Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation, Proceedings of the 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003. 87, 93, 94, 95

[Kur94] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994. 9, 10, 24, 94, 95

[KV01] O. Kupferman and M. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001. 2, 4, 20, 21, 22, 44, 68, 79, 84, 92, 102, 112

[KVW00] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000. 9

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. 1, 2

[Lam80] L. Lamport. “Sometime” is sometimes “not never” — on the temporal logic of programs. In POPL'80 [POP80], pages 174–185. 7, 8

[Lam83] L. Lamport. What good is temporal logic? In R. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 657–668. North-Holland/IFIP, 1983. 2, 20

[Lam85] L. Lamport. Logical foundation. In M. Paul and H. Siegert, editors, *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985, Munich*, volume 190 of *LNCS*, pages 19–30. Springer, 1985. 20

[Lam04] L. Lamport. Liveness manifesto. In Podelski et al. [PSZ]. 1, 2

[Lar04] K. Larsen. Liveness manifesto. In Podelski et al. [PSZ]. 1

[Lat03] T. Latvala. Efficient model checking of safety properties. In T. Ball and S. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *LNCS*, pages 74–88. Springer, 2003. 84

[LBHJ04] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple bounded LTL model checking. In Hu and Martin [HM04], pages 186–200. 24

[LBHJ05] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple is better: Efficient bounded model checking for past LTL. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17–19, 2005, Proceedings*, volume 3385 of *LNCS*, pages 380–395. Springer, 2005. 5, 8, 24, 45, 74, 82, 84, 98, 99, 100, 102, 112, 115

[LH00] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundam. Inform.*, 43(1-4):175–193, 2000. 45

[LMS02] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 383–392. IEEE Computer Society, 2002. 8, 12, 17, 74

[LNA99] J. Lind-Nielsen and H. Andersen. Stepwise CTL model checking of state/event systems. In Halbwachs and Peled [HP99], pages 316–327. 94, 95

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'85), New Orleans, LA, USA, January 14-16, 1985*, pages 97–107. ACM Press, 1985. 8, 9, 83

[LPJ+96] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In Rutenbar and Otten [RO96], pages 76–81. 94, 95

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *STTT*, 1(1–2):134–152, 1997. 2, 47, 65

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, *Logic of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings*, volume 193 of *LNCS*, pages 196–218. Springer, 1985. 8, 18, 20

[LS06] B. Li and F. Somenzi. Efficient abstraction refinement in interpolation-based unbounded model checking. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 – April 2, 2006, Proceedings*, volume 3920 of *LNCS*, pages 227–241. Springer, 2006. 108

[Mai04] P. Maier. Intuitionistic LTL and a new characterization of safety and liveness. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *LNCS*, pages 295–309. Springer, 2004. 20

[McM] K. McMillan. Cadence SMV. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv. 4, 10, 100

[McM93] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993. 9, 10, 23, 28, 39, 83, 92, 94

[McM03] K. McMillan. Interpolation and SAT-based model checking. In Hunt Jr. and Somenzi [HS03], pages 1–13. 9, 113

[MOSS99] M. Müller-Olm, D. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In A. Cortesi and G. Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *LNCS*, pages 330–354. Springer, 1999. 12

[MP83] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83), Austin, TX, USA, January 24-26, 1983*, pages 141–154. ACM Press, 1983. 7

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Principles of Distributed Computing, Proceedings of the Ninth Annual ACM Symposium, PODC 1990, Quebec City, Quebec, Canada, August 22-24, 1990*, pages 377–410, 1990. 2, 20, 83

[MT03] P. Manolios and R. Trefler. A lattice-theoretic characterization of safety and liveness. In *Principles of Distributed Computing, Proceedings of the Twenty-Second Annual ACM Symposium, PODC 2003, Boston, MA, USA, July 13-16, 2003*, pages 325–333, 2003. 20

[MY01] T. Margaria and W. Yi, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *LNCS*. Springer, 2001. 133, 149

[NuS] The NuSMV Team. NuSMV. `http://nusmv.irst.itc.it/`. 100

[NW97] U. Nitsche and P. Wolper. Relative liveness and behavior abstraction (extended abstract). In *Principles of Distributed Computing, Proceedings of the Sixteenth Annual ACM Symposium, PODC 1997, Santa Barbara, CA, USA, August 21-24, 1997*, pages 45–52, 1997. 44

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Inf.*, 6:319–340, 1976. 8

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982. 2

[PBG05] M. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005. 9

[Pel96] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996. 10

[Pel01] D. Peled. *Software Reliability Methods*. Springer, 2001. 8, 9, 17, 20, 44

[Pix91] C. Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In Clarke and Kurshan [CK91], pages 54–64. 9

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77), Providence, RI, USA, October 31 – November 2, 1977*, pages 46–57. IEEE Computer Society, 1977. 7, 8

[Pod04] A. Podelski. Liveness manifesto. In Podelski et al. [PSZ]. 111

[POP80] *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80), Las Vegas, NV, USA, January 28-30, 1980*. ACM Press, 1980. 139, 142

[POP02] *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02), Portland, OR, USA, January 16-18, 2002*. ACM Press, 2002. 134, 140

[PR04] A. Podelski and A. Rybalchenko. Transition invariants. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 32–41. IEEE Computer Society, 2004. 65, 66

[PR05] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 132–144. ACM, 2005. 65, 66

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In Emerson and Sistla [ES00], pages 328–343. 2, 66

[PSZ] A. Podelski, B. Steffen, and L. Zuck, editors. *Liveness Manifestos. Beyond Safety, International Workshop, Schloß Ringberg, Germany, April 25–28, 2004*. http://www.cs.nyu.edu/acsys/beyond-safety/liveness.htm. 2, 111, 141, 143, 145, 149

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *LNCS*, pages 337–351. Springer, 1982. 8

[RBS00] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In Hunt Jr. and Johnson [HJ00], pages 143–160. 45

[RDH03] Robby, M. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, Helsinki, Finland, September 1-5, 2003*, pages 267–276. ACM, 2003. 10

[RO96] R. Rutenbar and R. Otten, editors. *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'96), San Jose, CA, USA, November 10–14, 1996*. IEEE Computer Society, 1996. 141, 143

[RS04] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In Jensen and Podelski [JP04], pages 31–45. 45, 115

[Sav70] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970. 45

[SB95] C. Seger and R. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995. 2

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In Emerson and Sistla [ES00], pages 248–263. 71, 83, 115

[SB03] V. Schuppan and A. Biere. Verifying the IEEE 1394 FireWire Tree Identify Protocol with SMV. *Formal Asp. Comput.*, 14(3):267–280, 2003. 5, 98, 99

[SB04] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *STTT*, 5(2-3):185–204, 2004. 5, 40, 42, 98

[SB05] V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of LTL with past. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *LNCS*, pages 493–509. Springer, 2005. 5

[SB06] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. In S. Smolka and J. Srba, editors, *Proceedings of the 7th International Workshop on Verification of Infinite-State Systems, INFINITY '05, San Francisco, CA, USA, August 27, 2005*, ENTCS, 149(1), pages 79–96. Elsevier, 2006. 5

[SC85] A. Sistla and E. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985. 8, 22, 44

[Sch01] K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *LNCS*, pages 39–54. Springer, 2001. 18, 83

[Sch02] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002. 58

[Sch03] T. Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 1–6. ACM, 2003. 3, 8

[Sht01] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In T. Margaria and T. Melham, editors, *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, volume 2144 of *LNCS*, pages 58–70. Springer, 2001. 24, 87

[Sis94] A. Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994. 2, 20

[SL88] F. Schneider and L. Lamport. On E. W. Dijsktra's position paper on "fairness:". *ACM SIGSOFT Software Engineering Notes*, 13(3):18–19, 1988. 1

[Som] F. Somenzi. Wring 1.1.0. `ftp://vlsi.colorado.edu/pub/Wring-1.1.0.tar.gz`. 36

[SS04] T. Schuele and K. Schneider. Bounded model checking of infinite state systems: Exploiting the automata hierarchy. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on. San Diego, CA, USA, June 23 – 25, 2004*, pages 17–26. IEEE, 2004. 83

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In Hunt Jr. and Johnson [HJ00], pages 108–125. 9, 93

[ST03] R. Sebastiani and S. Tonetta. "More deterministic"vs. "smaller"Büchi automata for efficient LTL model checking. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *LNCS*, pages 126–140. Springer, 2003. 84

[Sti96] C. Stirling. Games and modal mu-calculus. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *LNCS*, pages 298–312. Springer, 1996. 43

[SVW87] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theor. Comput. Sci.*, 49(2–3):217–237, 1987. 18, 115

[SY01] N. Shilov and K. Yi. On expressive and model checking power of propositional program logics. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers*, volume 2244 of *LNCS*, pages 39–46. Springer, 2001. 43

[SYE+05] N. Shilov, K. Yi, H. Eo, S. O, and K.-M. Choe. Proofs about folklore: why model checking = reachability? Submitted, 2005. 43, 65, 113

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. 18

[TH02] H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *STTT*, 4(1):57–70, 2002. 84

[Tho90] W. Thomas. Automata on infinite objects. In van Leeuwen [vL90], pages 133–192. 17, 18

[Tho97] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 389–455. Springer, 1997. 36

[UN02] U. Ultes-Nitsche. Do we need liveness? — Approximation of liveness properties by safety properties. In W. Grosky and F. Plasil, editors, *SOFSEM 2002: Theory and Practice of Informatics, 29th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 22-29, 2002, Proceedings*, volume 2540 of *LNCS*, pages 279–288. Springer, 2002. 44

[Val92] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992. 10

[Var01] M. Vardi. Branching vs. linear time: Final showdown. In Margaria and Yi [MY01], pages 1–22. 8

[Var04] M. Vardi. Liveness manifesto. In Podelski et al. [PSZ]. 2

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. 10

[VIS96] The VIS Group. VIS: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Computer Aided Verification, Proceedings of the 8th International Conference, CAV'96, New Brunswick, NJ, USA, July 31 – August 3, 1996*, volume 1102 of *LNCS*, pages 428–432. Springer, 1996. 10, 71, 98

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*. Elsevier and MIT Press, 1990. 137, 148

[VW86] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86), Cambridge, MA, June 16-18, 1986*, pages 332–344. IEEE Computer Society, 1986. 2, 3, 9, 21, 22, 83

[VW94] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994. 4, 18, 83, 115

[WB98] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In Hu and Vardi [HV98], pages 88–97. 4, 47, 112

[WKS01] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001* [DAC01], pages 542–545. 24, 87

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983. 18, 115

[WVS83] P. Wolper, M. Vardi, and A. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS'83), Tuscan, AZ, USA, November 7-9, 1983*, pages 185–194. IEEE Computer Society, 1983. 18, 83

[Yan] B. Yang. SMV models. `http://www.cs.cmu.edu/~bwolen/software/smv-models/`. 98, 99

[YS01] J. Yang and C. Seger. Introduction to generalized symbolic trajectory evaluation. In *19th International Conference on Computer Design (ICCD 2001), VLSI in Computers and Processors, 23-26 September 2001, Austin, TX, USA, Proceedings*, pages 360–367. IEEE Computer Society, 2001. 2

[zCh] The SAT Group at Princeton University. zChaff. `http://www.princeton.edu/~chaff/zchaff.html`. 100

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In R. Ernst, editor, *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'01), San Jose, CA, USA, November 4–8, 2001*, pages 279–285. IEEE Computer Society, 2001. 100

Curriculum Vitae

Viktor Schuppan

September 5, 1973	Born in München, Germany
1980 – 1984	Primary school, Putzbrunn
1984 – 1993	Gymnasium Neubiberg
1993	Abitur
1993 – 1999	Studies in Computer Science, TU München
1995 – 1996	Erasmus year at Queen's University Belfast, UK
1996	Internship at Siemens, München
1997	Internship at HYPO-Bank, München
1999	Diploma in Computer Science, TU München
1999 – 2000	Alternative national service, Klinikum Großhadern
since 2001	Research and Teaching Assistant Computer Systems Institute, ETH Zurich