System Verification Tools Based on Monadic Logics

Dissertation
zur Erlangung des Doktorgrades
der Fakultät für Angewandte Wissenschaften
der Albert-Ludwigs-Universität Freiburg im Breisgau
vorgelegt von

Abdelwaheb Ayari

Im Februar 2003

Dekan:	Prof. Dr. Thomas Ottmann
Erstreferent:	Prof. Dr. David Basin
Zweitreferent:	Prof. Dr. Javier Esparza

إقرأ باسم ربك الذي خلق
خلق الإنسان من علق
إقرأ وربك الأكرم
الذي علّم بالقلم
علّم الإنسان ما لم يعلم

Proclaim! (or Read!) In the name of thy Lord and Proclaim!
(or Read!) In the name of thy Lord and Cherisher, Who created,
Created man, out of a (mere) clot of congealed blood:
Proclaim! and thy Lord is Most Bountiful,
He Who taught (the use of) the Pen,
Taught man that which he knew not.

Holy Quran Surat Al-Alaq (1-5)

to my parents & Ahlem

Zusammenfassung

In der Mitte des letzten Jahrhunderts erschienen die ersten Arbeiten über monadische Logiken zweiter Stufe. Das Interesse an diesen Logiken lag zunächst hauptsächlich an Entscheidbarkeitsfragen von arithmetischen Theorien. Die monadischen Logiken zweiter Stufe über Wörter und Bäume gehören zu den ausdrucksstärksten Logiken, die noch entscheidbar sind. Gegenwärtig werden monadische Logiken auch in der Informatik zum Zweck der formalen Systemverifikation verwendet. Entscheidungsverfahren für diese Logiken wurden in verschiedenen Werkzeugen wie z.B. MONA, MOSEL und dem STEP System implementiert und teilweise erfolgreich in unterschiedlichen Anwendungsgebieten, vor allem in der Hardware- und Protokollverifikation, eingesetzt.

Der Erfolg der auf monadischen Logiken basierten Verifikationswerkzeuge wird allerdings durch zwei große Nachteile, die diese Logiken mit sich bringen, erheblich vermindert. Zum einen sind diese Logiken wegen ihres geringen Abstraktionsgrades als Spezifikationssprachen ungeeignet; die Formalisierung von Systemen und Systemeigenschaften in diesen Logiken bedarf eines hohen Maßes an Erfahrung und Detailkenntnissen und ist mit der Programmierung in Assembler vergleichbar. Zum anderen haben die Entscheidungsverfahren für diese Logiken eine sehr hohe Berechnungskomplexität; oft brechen die oben genannten Werkzeuge ihre Berechnung aus Mangel an Speicherressourcen ab.

Die vorliegende Arbeit stellt mehrere Verfahren vor, die die erwähnten Nachteile überwinden und machen somit die monadischen Logiken für die Praxis besser nutzbar. Nachfolgend stellen wir die Beiträge in unserer Arbeit dar. Wir entwickeln zunächst eine neue auf monadische Logik über endliche Bäume basierende Spezifikationssprache, die intuitiv und benutzerfreundlich ist und Sprachkonstrukte bereitstellt, die einen höheren Abstraktionsgrad ermöglichen. Außerdem geben wir eine syntaktische Charakterisierung von Klassen von Formeln der neu entwickelten Spezifikationssprache an, die eine akzeptable Berechungskomplexität haben.

Desweiteren untersuchen wir das Problem der sogenannten Zustandsraumexplosion: Bei der Verifikation von großen Systemen in monadischen Logiken kann der Zustandsraum nicht-elementar groß werden. Um dieses Problem zu vermeiden, geben wir ein Verfahren an, das für die Generierung von Gegenbeispielen eine effektive und nicht-elementare Verbesserung gegenüber den herkömmlichen Entscheidungsverfahren bietet.

Schließlich beschäftigen wir uns auch mit der Frage, wie man, ausgehend von der Kernidee dieser Methode zur Generierung von Gegenbeispielen, monadische Logiken über endlichen Wörtern zum Nachweis von Eigenschaften nicht-terminierender Systeme benutzen kann. Unsere Resultate ergeben, daß man sowohl Sicherheits- als auch Lebendigkeitseigenschaften in monadischen Logiken über endlichen Wörtern formalisieren und dadurch automatisch beweisen kann.

Die Praxistauglichkeit unserer theoretischen Resultate stellen wir durch die Implementierung von verschiedenen Verifikationswerkzeugen (LISA, MONACO und QUBOS) unter Beweis. Anwendbarkeit und Skalierbarkeit dieser Werkzeuge werden anhand nicht-trivialer Fallbeispiele evaluiert.

Preface

Fundamental work on monadic second-order logics began about forty years ago. These logics are amongst the most expressive logics that are known to be decidable. Their first application domain was mathematics, where they were for example used to decide theories of arithmetics. Recently, they have also been applied to formally reason about a number of problems in computer science: despite their non-elementary complexity, decision procedures for monadic logics over finite words and finite trees have been implemented in numerous tools (*e.g.* MONA, MOSEL, and STEP) and have been successfully applied to problems such as the verification of hardware and software systems. These logics suffer, however, from two drawbacks that strongly limit their application, namely the low-level language they provide to specify systems and properties, and the demanding computational complexity of their decision procedures.

To make system verification based on monadic logics more viable in practice, in this thesis we systematically address both these problems at once. To this end, we first improve the existing approaches (i) by formalizing a new specification language which is expressive, intuitive and more user-friendly, and (ii) by providing a handle on the complexity of the logics' decision procedures. Second, we develop new efficient algorithms and approaches to cope with the state-space explosion problem. Third, we investigate how to employ the monadic logic over finite words to reason about non-terminating systems. Finally, we implement our methods in three tools (LISA, MONACO, and QUBOS) and show their applicability and scalability.

Acknowledgments

I owe much to my supervisor, David Basin. He has not only educated me to think for myself, but also supported and encouraged me when science did not. He has always found the time and patience to listen to my intellectual ramblings.

My roommates Stefan Friedrich and Sebastian Mödersheim deserve a particular mention for tolerating my many moods and for our discussions on many subjects.

Over the time I spent for my thesis in Freiburg, I had the luck to be within an excellent family: Ina Eckert, Erika Abraham-Mumm, Luca Viganò, Rafael Accorsi, Martin Preen, Peter Leven, Felix Klaedtke, Jürgen Doser, Frank Rittinger, Achim Dominik Brucker, Torsten Lodderstedt, Burkhart Wolff, Jan-Georg Smaus, Georg Struth, Olga Levchuk, Peter Thiemann, Simon Helsen, Matthias Neubauer, and Ulf Schünemann.

I thank Mongi and Mounira Kasmi, Hamadi and Schadlia Naffati, Saleh and Hassna Saidani, and the sunday soccer group: Thomas, Walter, Heinz, Rudi, Wilhelm, Martin, and Imed.

My family I cannot thank enough. Without their unconditional and lasting support, a lifetime long, I would not be writing any of this.

Freiburg, February 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we provide an overview of the thesis. We begin by motivating our work and proceed with a presentation of its main goals and results. Finally, we give an outline of the subsequent chapters.

1.1 Motivation and Scope

In the last two decades, computer systems have become a fundamental component in almost all fields of everyday life. The use of these systems ranges from mundane applications like computer games to highly critical applications such as the control of nuclear power plants. Whereas failures in mundane systems lead to the frustration of users, failures in critical systems may have disastrous consequences. For instance, failures in medical instruments could lead to the death of patients, failures in electronic commerce systems could lead to the bankrupt of financial institutes, and failures in the guidance systems of planes could lead to human casualties.

Hence, for all systems used in critical and life-threatening applications, *i.e.* for safety-critical systems, failures are inadmissible. In order to avoid these errors, one should ensure that all possible behaviors of a system, and in particular of a safety-critical system, are faultless. This task is called *system verification* and work in this area was already initiated in the late 60's by Floyd [Flo67] and Hoare [Hoa69].

The development of reliable and correct hardware and software systems is still a main challenge in computer science. Besides for the dangerous consequences of faulty systems, the costs of correcting errors discovered during the design phase of systems are much smaller than the costs needed for correcting errors of systems that are already deployed. These economic reasons have encouraged the industry to collaborate with research institutes and to sponsor numerous projects with the objective of developing methodologies that support the construction of correct systems.

In the following, we discuss two approaches that are used to tackle the problem of establishing the correctness of systems. The first approach is *simulation and testing.* It has a widespread use in the industry because it is simple and well-understood. The second approach is *formal methods.* This approach is presently less used in the industry than the previous approach because it requires thorough knowledge of mathematics and logics on the part of engineers and increases significantly the costs of the system development.

Simulation and Testing

Conventional methods for simulation and testing allow for the validation of some selected behaviors of a system. Testing is performed on systems before their deployment and it consists of calling systems with selectively generated inputs for system parameters like buffer sizes, list lengths, etc. Simulation is similar to testing, however, it is performed on abstract models of actual systems.

In order to increase the chance to uncover errors, *benchmarks* are built by making "clever/educated guessing" of concrete input values of the parameters such that a systematical testing of all parts of the system under consideration is possible. The programming of benchmarks is, however, for large and unintuitive systems a non-trivial and time consuming task and thus, to keep the costs of system development minimal, in practice, the concrete input values for the parameters are generated randomly. While simpler and faster to set up, random testing is less successful than testing using clever benchmarks.

For systems with a finite (and small) number of behaviors simulation and testing can be exhaustive. They are in this case *complete*, *i.e.* if an error exists in a system than it can be uncovered. Since systems generally have a very large (or even infinite) number of possible behaviors, simulation and testing results tend to become impracticable. As already stated by Dijkstra [Dij72], the classical simulation and testing methods can only prove the presence not the absence of failures in systems, and consequently, they fail to establish the correctness of systems.

Formal Methods

Formal methods, in contrast to conventional testing and simulation, offer an appropriate framework in which safety critical systems can be investigated for correctness. Formal methods are mathematical notations and techniques designed to specify system properties in an unambiguous way and to carry out mathematical proofs that the specified properties are satisfied by the system. Depending on the degree of proof automation, we distinguish three approaches to system verification.

The first approach is called *interactive theorem proving* and is usually based on very expressive undecidable logics such as *higher-order logic*. In this approach theorem provers provide specification languages in which system properties, as well as system descriptions, can be formalized. In addition, they provide a logical deduction systems where proofs are constructed under interactive user guidance. The expressive power of the underlying logic reflects the high-level nature of the provided specification language. Generally, with such specification languages, problems can be formalized in a fairly adequate and natural way. In interactive theorem proving tools, proof automation is usually provided to some extent, *e.g.* in the form of proof tactics. However, great skill is required on the part of the software engineer to guide the proof search process. Examples of successful and widely used interactive theorem provers are ISABELLE [Pau94], PVS [ORRS96], and HOL [Gor93].

The second approach is called *automatic theorem proving* and is based on semi-decidable logics such as *first-order logic*. Automatic theorem provers are often referred to as "push-button" tools, which indicates that they are easy to use and do not require any special background in proof construction, since no interaction of the user with the proof system is required at all. Nevertheless, some familiarity with logic is still needed in order to model systems and formalize properties within the logic under consideration. In these tools, it is not guaranteed that the proof search process always terminates. Examples of automatic theorem provers are OTTER [MW97] and SPASS [WAB$^+$99].

The third approach is called *complete theorem proving* and this is the approach we are dealing with in this dissertation. Complete theorem proving sacrifices expressiveness in favor of complete automation, that is, the problem of automatic verification is tackled by employing less expressive logics than in the previous two approaches, and for which decision procedures exist. The decision procedures form the kernel of the deduction mechanism of such provers.

The most beneficial advantage of many complete theorem provers is that it can often be easily equipped with a counter-example mechanism, *i.e.* if the system property to be checked is not valid, then an example demonstrating how the formula can be falsified is automatically generated. In terms of system verification, this means that the provers are able to generate system behaviors that violate the given specification. Such faulty behaviors are used afterwards to debug the system and to localize the failure. The main drawback of this approach is its limited scope of applicability and this of course is not surprising because of the tradeoff between expressiveness, *i.e.* what systems and properties can be formalized and analyzed, and the amount of automation, *i.e.* to which extent the verification can be performed automatically.

In the following, we discuss two kinds of system verification methods based on automatic theorem proving and illustrate the scope of our dissertation.

Model Checking

Model checking[1] is a special kind of automatic verification method. While the verification problem in theorem proving consists of proving the validity of a given formula, *i.e.* proving the satisfiability of the formula in all interpretation domains, the verification problem in model checking consists of proving the satisfiability of a given formula for a specific interpretation domain. Moreover, the interpretation domains in model checking correspond to finite models of systems. Typical systems here are concurrent finite systems, digital circuits and communication protocols; more generally, systems whose intricacy resides in control rather than in data. Model checking can also be applied to systems with infinite number of states and in this case preprocessing techniques like abstraction [CGL94] are used to extract systems with finite numbers of states.

In practice, we generally distinguish two approaches to model checking. The first approach is called *temporal logic model checking* [EE81, QS82]. In this approach systems are modeled as finite transition systems and system properties are expressed using temporal logics which are kinds of decidable modal logics such as the Linear Temporal Logic (LTL [Pnu77]) and the Branching Time Temporal Logic (CTL [BAMP81, EE81]). The verification algorithms in this approach differ mainly in the state representation of the transition graphs. The model checkers of the early 80's [HK91, BCDM86, CES86] used an explicit state representation of the transition graphs in their implementations. The verification task is carried out by an exhaustive traversal search through the reachable state-space and by the validation of the specification in each state. The use of these model checkers allowed for the automatic discovery of nontrivial errors in circuits and protocols of small size. Later, in the early 90's, work on *Binary Decision Diagrams* (BDDs) [BRB90] laid the foundation for a new generation of model checkers that use an implicit (*symbolical*) representation of the transition graphs. The transition graph is represented by Boolean formula using BDDs and the verification task is then a fixed-point computation of predicate transforms that are extracted from the specification to be checked and the transition graph [McM92]. The use of BDDs has increased the scope of verification capabilities of model checkers, as it is demonstrated, using the SMV [McM92] model checker, by the verification of the cache consistency protocols for the Encore Gigamax [MS91] and the IEEE Futurebus+ standard [CGH$^+$93].

The second approach to model checking is based on automata theory. The system to be modeled is converted into a finite automaton that exactly accepts the behaviors of the system, and the system property to be checked, expressed often in LTL, is negated and then translated into finite automaton that accepts exactly the forbidden behaviors of the system. The verification task is then reduced to the

[1]In this thesis, we focus only on model checking of finite-state systems.

automata-theoretic problem of checking the emptiness of the intersection automaton of the two automata [HK90, Kur94]. There are also two different ways to accomplish the verification task: one way consists of constructing the intersection automaton and checking thereafter its emptiness using graph search algorithms. Another way, called *on-the-fly model checking*, consists of building the intersection automaton incrementally: a new portion of state-space is constructed only if no counter-example to the property is detected. So, the construction of the intersection is stopped when a counter-example is found. The on-the-fly tactic is adopt by the Spin model checker [Hol97] which was used in many successful applications.

Despite several success stories of model checking in practice, there are limitations on the use of this verification method in terms of the size of the system. It is often the case that model checkers are faced with the necessity to construct far too many states, a problem referred to as the *state-space explosion problem*. In recent years, many researchers have been developing promising techniques to cope with the state-space explosion problem, *e.g.* abstraction [CGL94], symmetry [CJEF96, ES93, ID93], partial-order reduction [GP93, Pel94], and bounded model checking [BCCZ99].

System Verification Based on Monadic Second-order Logics

Monadic second-order logic has an extremely simple syntax: formulae are constructed from first-order and second-order variables (monadic predicates), successor relations and are closed under Boolean connectives and quantification over first-order and second-order variables. There are several interpretation domains for the monadic second-order logic and we will only focus on those logics that are interpreted over words and trees. More precisely, we consider the monadic second-order logics.

- over words (also called *monadic second-order logics of one successor*)
  - M2L-Str and WS1S over finite words, and
  - S1S over infinite words,
- over trees (also called *monadic second-order logics of two successors*)
  - M2L-Tree and WS2S over finite trees, and
  - S2S over infinite trees.

Historically, research on monadic second-order logics over words has been triggered by work on decision problems for weak systems of arithmetic and by work on the description of the dynamic behavior of non-terminating circuits [Chu62, TB73]. About forty years ago, Büchi [Büc60], Elgot [Elg61], and Trakhtenbrot [TB73] already formalized the connection between finite automata on finite words and monadic logics over finite words. They proved that the expressive power of these logics and that of regular languages coincide, and they gave an effective procedure to build finite automata on finite words from formulae and vice versa. Büchi [Büc62] and McNaughton [McN66] have shown that the same connection holds between ω-regular languages and the monadic logics over infinite words, and described how Büchi automata can be built from monadic formulae and vice versa.

The monadic logics over trees are a generalization of the monadic logics over words. Doner [Don70] and Thatcher and Wright [TW68] showed that these logics have the same expressive power as regular tree languages. For this purpose, they used finite automata on trees. Rabin [Rab69] showed the same equivalence between the monadic logics over infinite trees and the ω-regular tree languages.

The finite automaton for a formula of these logics is constructed inductively over the structure of the formula. The logical connectives for negation, disjunction, and conjunction correspond to the automata-theoretic operations complementation, union, and product. The quantification over monadic predicates (the other kinds of quantification are inessential, as they can be converted into monadic quantifications) corresponds to the automata-theoretic projection operation. A formula is valid if and only if the associated automaton accepts all the words[2] over a given alphabet. If the formula is not valid, then the associated automaton can be analyzed for satisfying examples as well as counter-examples. Meyer [Mey75] showed that the transformation of a formula into an automaton requires, in the worst case, non-elementary space and time. That is, the minimal automaton representing a formula of size n may require space whose lower bound is an iterated stack of exponentials whose height is proportional to n.

Despite their atrocious complexity, the decision procedures for monadic logics over finite words and finite trees have been implemented in numerous tools, *e.g.* MONA [KM01], Mosel [MC97], MOSEL[3] [KMMG97], and the STEP system [MBBC95], and have been successfully applied to problems in diverse domains, including hardware [BK98] and protocol [HJJ+96] verification. The MONA system is one of the best-known and widely used tools. MONA follows the automata construction described above: it translates formulae into minimal deterministic finite automata whereby it uses BDDs [Bry86, Bry92] for a compressed representation of the transition function of automata.

The verification of a system *Sys* with respect to a property *Spec* is typically performed as follows: first, the system *Sys* and the property *Spec* are encoded as monadic second-order formulae Φ_{Sys} and Φ_{Spec} ,respectively. Second, the determinis-

[2]Models in monadic second-order logics over words are identified with words.

[3]This tool and the previously mentioned one have up to lowercase/uppercase the same names. They were, however, independently developed at the university of Tübingen and the university of Dortmund, respectively.

tic finite automaton corresponding to the implication $\Phi_{Sys} \rightarrow \Phi_{Spec}$ over the alphabet $\mathbb{B}^n$, where n is the number of the free variables occurring in Φ_{Sys} and Φ_{Spec}, is constructed. Finally, the automaton is analyzed: if it accepts all words then the above implication is valid which means that the system behaves correctly with respect to the specification. If the automaton does not accept all words over $\mathbb{B}^n$, then there is some word representing a possible behavior of the system that does not satisfy the specification. In this case, two words (examples) of minimal length can be generated: one of them satisfies the specification[4] and the other one violates the specification.

From a theoretical point of view, the system verification approach based on monadic logics is superior to the temporal logic model checking approach at least in two aspects. First, the above presented monadic logics are more expressive than the temporal logics LTL and CTL. This means that the class of systems and system properties that can be formulated and analyzed using monadic logics is larger than the class of systems and properties that can be analyzed by LTL and CTL model checking. Moreover, the monadic logics are non-elementary more succinct, that is, some system descriptions and properties can be expressed as monadic formulae that are non-elementary shorter than their equivalent temporal formulae. Second, in contrast to temporal logics, in monadic logics we have a parameter that can be used to explicitly reason about points and intervals of time, data paths in circuits, or numbers of agents in protocols. This allows for the analysis of parameterized systems like n-bit adders and counters.

From the practical side, model checking is less resource intensive than system verification based on monadic logics. The LTL model checking problem for example is *PSPACE-complete* [SC85] and the CTL model checking problem is in deterministic polynomial time [Wol86], whereas the monadic logics are non-elementary decidable. In this thesis, we address this practical deficiency of monadic logics.

1.2 Goals

The overall goal of this thesis is to make system verification based on monadic logics more usable in practice. This includes:

(1) The improvement of existing approaches in two respects: (i) to extend specification languages with high-level features that make these languages more user-friendly, and (ii) to provide a handle on the computational complexity of the decision procedures.

(2) The development of new efficient algorithms and new approaches to cope with the state-space explosion problem.

[4]We assume that the specification Φ_{Spec} is satisfiable, which can be separately checked.

(3) The investigation of reasoning about non-terminating finite systems using monadic logics over finite words.

(4) We show the applicability of system verification based on monadic logics on non-trivial examples.

In the following, we illustrate these objectives in more detail.

A High-Level Specification Language with a Complexity Estimation Mechanism

In the verification approach based on monadic logics, system descriptions and system properties are both formulated directly within the language of the underlying logic. These "specification languages" usually do not provide any high-level programming concepts that help the user to model application problems in a structured and comprehensible way. From our experience, we can best compare these languages with assembly languages, where users often struggle with painful encoding tasks.

On the other hand, it is a fact of life that the step from informal towards formal requirements (this step is also called *requirements definition* in the software development process) cannot be formally checked: it is not possible to prove in a mathematical way that the informal requirements that one may have in mind are captured by some formal specification formulated in some logic. In this phase of the software development process, one traditionally trusts the developer's familiarity with formal specifications as well as the abstraction power that a specification language offers to model requirements in an adequate way. If the specification language lacks high-level programming notations, the gap between the informal and formal sides is even larger. The bigger the systems to be modeled are, the more dramatic the situation becomes. We think that in such situations the confidence in formal, but incomprehensible, specification is low and thus, proving correctness of systems is of arguable value. Another problem which appears with unstructured specification is that the user has no control over the computational complexity of the verification task. This means that the user cannot estimate the size of the automaton that has to be constructed and therefore the running time of the decision procedure is unpredictable. More importantly, the user often has no support to organize the specification in such a way that the computation costs is still admissible and realistic.

Experience indicates that, although tools like MONA, Mosel, and the others mentioned above, are powerful aids to verification, their usefulness is limited by the two aforementioned problems [BK98]. Our aim is therefore to develop a specification language on top of WS2S, which, similar to high-level programming languages, provides high-level programming primitives that allow a structured specification in a natural and comprehensible way. Additionally, the specification language should offer means that can be used to control the complexity of the verification problem under consideration.

The State-Space Explosion Problem

Although verification tools based on monadic logics have been successfully applied to an interesting class of problems, there is still a large class of problems for which most tools abort the automata construction due to limited space and therefore fail to achieve the verification task. This is not surprising since for example for concurrent systems with large number of components we have to construct automata with very large number of states. In other words, we have to deal with the state-space explosion problem. This is analogous to state-space explosion in model checking where the state-space is exponential in the number of Boolean state variables, except that for monadic logics the number of states in the constructed automaton can be non-elementary in the size of the input formula! Besides the state-space explosion problem, there is an additional problem stemming from the internal representation of the automata. Like the system Mona, most other tools represent the transition function of automata using BDDs, which become too big and unmanageable when the variable ordering is unfavorable.

In order to cope with the state-space explosion problem, we aim to develop new techniques and algorithms, not necessarily based on BDDs, that allow the practical verification of large systems. We investigate the question of whether it is possible to develop efficient (elementary instead of non-elementary) verification methods by weakening the properties to be checked. We restrict ourselves to monadic logics over words as well as to the case where for a given formula and a natural number n, there are (counter-)models of length n for the formula.

Expressing Liveness Properties in M2L-Str and WS1S

For non-terminating concurrent systems, one usually distinguishes between two kinds of properties: safety and liveness [AS85]. Safety formulae are properties of infinite system computations (behaviors) that state that some "bad thing" does not happen in the computation. A safety property does not hold for a computation if a finite prefix of the computation contains a position where some "bad thing" happens, and thus safety properties can be formalized and analyzed using the monadic logics over finite words, M2L-Str and WS1S. Liveness formulae are properties of infinite computations that state that a "good thing" happens in the computation. A liveness property cannot be checked using finite prefixes of computations, as the required "good things" can still occur in the rest of the computations. Thus, the formulation of liveness properties in monadic logics over finite words appears a priori impossible. The S1S logic on the one hand, permits the formulation of liveness properties, but on the other hand is intractable. Indeed, there is no efficient implementation of the decision procedure for this logic.

Our aim here is to investigate if monadic logics over finite words can be used to express liveness properties.

Applicability and Scalability of our Approaches

Our work does not only focus on the theoretical side, but also highlights some aspects of the practical side. We find it important to evaluate our ideas and contributions by implementing our approaches in concrete tools and then showing their usability, applicability, and scalability. Our goal is to provide tools that can be effectively employed in different phases of the system development process to ensure the correctness of software and hardware systems.

1.3 Main Results

The main results that we have established can be summarized as follows.

Lisa

We developed a new specification language, Lisa, built on top of the weak monadic second-order logic over finite trees and based on feature trees. Lisa provides a means for abstractly formalizing data using type declarations. Types structure the specification and interact with defined predicates by restricting the scope of quantification to elements of the defined types. Through the use of types, Lisa provides a new way for specifiers to estimate the complexity of their specifications.

Bounded Model Construction

In order to tackle the state-space explosion problem, we have explored the possibility of providing more efficient alternatives for counter-example generation than using standard automata-theoretic decision procedures for monadic logics over words. The problem is, given a monadic formula ϕ and a natural number k, to determine if ϕ has a word model of length k. Since we are concerned with constructing bounded models for formulae we call our problem *bounded model construction*, or BMC for short.

For M2L-STR, we have obtained a positive result: we can generate a formula in quantified Boolean logic (QBL) that is satisfiable if and only if ϕ has a word model of length k. The formula generated is polynomial in the size of ϕ and k, and can be tested for satisfiability in polynomial space.

We also have investigated bounded model construction for other monadic logics and established negative results. For WS1S we show that BMC is as hard as checking validity, which is non-elementary. This result is somewhat surprising since WS1S has the same expressive power and complexity as M2L-STR and their decision procedures differ only slightly. Our investigations showed that at least for counter-example generation, M2L-STR is the superior choice. Moreover, based on all these results we solved the open question whether the logic WS1S can be encoded in a validity-preserving way in the logic M2L-STR in elementary space and time. We proved that there is no such elementary encoding.

Reasoning about Non-terminating Systems

Using M2L-STR, we also showed how systems with infinite behaviors can be verified. This seems a priori impossible since this logic handles finite behaviors (words). We succeeded to embed LTL model checking in M2L-STR and demonstrated that we can use finite automata on finite words instead of Büchi automata. This embedding sets the basis for a comparison of LTL bounded model checking and bounded model construction for M2L-STR.

The LISA, MONACO, and QUBOS Tools

Motivated by our positive theoretical results, we have evaluated our ideas by implementing them and carrying out empirical tests.

The specification language LISA is implemented in a prototypical system that we also call LISA. It is currently coupled with the MONA system. The LISA system takes as input a program written in the LISA syntax and produces code in WS2S or in WS1S, which is then processed by MONA. We evaluated the LISA system by considering several case studies and the first results are very promising.

We have implemented the bounded model construction for M2L-STR in a system that we call MONACO. This system takes an M2L-STR formula and a bound (natural number) k and produces a quantified Boolean formula whose validity is then decided using a QBL solver. We have compared MONACO with the MONA system and our results document in most applications the superiority of MONACO. With this new tool, we are able to analyze very large systems expressed in the monadic logic M2L-STR.

We have developed QUBOS, a satisfiability solver for quantified Boolean Logic.

This work arose from our work on bounded model construction. First, we have defined a useful structural property of quantified Boolean formulae based on a notion of quantifier scope and have identified domains such as bounded model construction where formulae exhibit this natural class. Second, we have described how standard techniques like simplification and miniscoping can be combined to exploit the structure present in these formulae. Third, we have demonstrated empirically that our tool outperforms other state-of-the-art QBL solvers.

1.4 Dissertation Outline

The rest of this thesis is structured as follows:

Chapter 2 presents the formal background needed in the following chapters. Most of the terminology we use is standard, nevertheless, in order to keep this work as self-contained as possible, we have tried to provide all the definitions using mostly our own notation. In this chapter, we describe the relationship between regular languages (Section 2.2.1), finite automata (Sections 2.2.2-2.2.4), and monadic second-order logics (Sections 2.4-2.5). We discuss their expressive power and computational complexity. We also introduce the Mona system (Section 2.6) and show by means of examples how system verification is performed using Mona (Section 2.7).

Chapter 3 introduces the Lisa specification language. We define the kernel language (Section 3.2) and the type system (Section 3.4) and we provide a compilation of the Lisa kernel into WS2S (Section 3.3) as well as a compilation of the Lisa types into bottom-up deterministic tree automata (Section 3.4.2). We illustrate the use of Lisa by an example (Section 3.4.4). We treat the special case of Lisa where the trees resemble words (Section 3.6) and report on related work (Section 3.7).

Chapter 4 formulates the bounded model construction for several monadic second-order logics on finite words (Section 4.2) as well as on infinite words (Section 4.3). For the case of finite words, a positive result is established for the logic M2L-Str (Section 4.2.1) and a negative complexity result is established for the logic WS1S (Section 4.2.2). For the case of infinite words, a negative result is established for the logic S1S (Section 4.3.1). Further results are established for first-order fragments of S1S (Section 4.3.2 and Section 4.3.3).

Chapter 5 deals with the embedding of the LTL model checking problem in M2L-Str. We show how Büchi automata, finite-state systems, and LTL can be encoded in M2L-Str (Section 5.3). We prove then that both the validity problem and the model checking problem for LTL can be decided using the Mona system. Based on these results, we show that the bounded model constructor for M2L-Str can be used without loss of efficiency as a bounded model checker for LTL (Section 5.4).

Chapter 6 describes the MonaCo system which implements the bounded model construction approach for M2L-Str (Section 6.2). We report on several applications from various domains (Section 6.3).

Chapter 7 introduces the quantified Boolean solver Qubos. This solver is used by the MonaCo system. Although many solvers for quantified Boolean logic already exist, in this chapter we motivate the need for providing a new solver for this logic. We describe the problems that we intend to solve with Qubos and we explain which certain kinds of structure are present in these problems (Sections 7.2-7.3). We formalize a notion of structure based on relative quantifier scope and define how the structure in a problem can be measured. We describe (Section 7.4) the Qubos system and present the techniques used in it. Finally, we compare our system with other state-of-the-art solvers (Section 7.5).

Chapter 8 is the concluding chapter which summarizes our work and in which we discuss some goals for further research.

Chapter 2

Foundations

The subjects of this chapter are the mathematical concepts and notation used throughout this thesis. These concepts essentially include the theory of regular languages, finite automata, and monadic second-order logics. We will generally follow standard text books [Str94, UAH74] and the article [Tho90]. However, in some parts of our exposition we will deviate from the standards and we use our own terminology.

2.1 Organization

The chapter is organized as follows. In Section 2.2, we introduce regular languages and finite automata and briefly review the definitions of regular expressions, automata on finite words, Büchi automata, and automata on finite trees. In Section 2.4, we introduce several monadic second-order logics of one successor and their straightforward generalizations to logics of multiple successors. We explain in Section 2.4.5 (respectively Section 2.5.3) in which sense the logics of one successor (respectively multiple successors) are logics on words (respectively trees). In Section 2.6, we describe the MONA system, which implements the decision procedures of some of the aforementioned monadic second-order logics and demonstrate by an example how system verification is performed using monadic logics. Finally, Section 2.8 summarizes this chapter, and addresses drawbacks of system verification using MONA.

2.2 Regular Languages and Finite Automata

Let Σ be a finite nonempty set of symbols and $\mathbb{N}$ be the natural numbers. For $n \in \mathbb{N}$, we write $[n]$ for the set $\{0, \ldots, n-1\}$. For a set S, we denote by $\mathsf{Pow}(S)$ the set of all subsets of S.

Words A *finite word* over Σ of *length* n is a function w from $[n]$ to Σ. We use the notation $|w|$ to denote the length of w and w_i, instead of $w(i)$, to denote the symbol occurring in w at position i and we write the sequence $w_0, \ldots, w_{n-1}$ conventionally for w. The *empty* word, *i.e.* the function from $\emptyset$ to Σ, is denoted by ϵ. The concatenation of two words $a_0, \ldots, a_{m-1}$ and $b_0, \ldots, b_{n-1}$ is the word $a_0, \ldots, a_{m-1}, b_0, \ldots, b_{n-1}$. With Σ^n we identify the set of all words of length n, the Kleene-star operator $*$ is defined by $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$, and Σ^+ is defined as the set $\Sigma^* \setminus \{\epsilon\}$. A *language* L over Σ is a subset of Σ^*. The *concatenation* of two languages L_1 and L_2 is $L_1 L_2 = \{uv \mid u \in L_1,\, v \in L_2\}$. For $L \subseteq \Sigma^*$, L^* is the set $\{u \in \Sigma^* \mid u = x_0 x_1 \ldots x_n$ with $x_i \in L$ for $0 \leq i \leq n\}$. The *right-quotient* of a language L with a language L' is defined by $L/L' = \{w \mid$ there is $u \in L'$ such that $wu \in L\}$.

An *infinite word* over Σ is a function w from $\mathbb{N}$ to Σ. We write the infinite sequence $w_0, \ldots, w_{n-1}, \ldots$ for w. With Σ^ω we denote the set of all infinite words over Σ. Subsets of Σ^ω are called ω-languages. For $L \subseteq \Sigma^*$, L^ω is the set $\{\alpha \in \Sigma^\omega \mid \alpha = x_0 x_1 \ldots$ with $x_i \in L$ for $i \geq 0\}$.

For the (ω-)languages L_1 and L_2, the set operations $L_1 \cap L_2$, $L_1 \cup L_2$, and $L_1 \setminus L_2$, stand for intersection, union, and set difference respectively. The language $\overline{L}$ stands for the complement of L, which is $\Sigma^* \setminus L$, if $L \subseteq \Sigma^*$ and $\Sigma^\omega \setminus L$, if $L \subseteq \Sigma^\omega$

2.2.1 Regular Languages

Regular expressions and the languages they describe are defined inductively as follows:

- $\emptyset$, ϵ, and a are regular expressions and describe respectively the empty language $\emptyset$, the language containing the empty word $\{\epsilon\}$, and the singletons $\{a\}$ for each $a \in \Sigma$.

- If e_1 and e_2 regular expressions describing respectively the languages L_1 and L_2, then $e_1 + e_2$, $e_1 e_2$, e_1^*, and $\overline{e_1}$ are regular expressions that describe respectively the languages $L_1 \cup L_2$, $L_1 L_2$, L_1^*, and $\overline{L_1}$.

A *regular language* is a language described by a regular expression. Regular expressions built without the star operation $*$ are called *star-free regular expressions* and they describe *star-free regular languages*, which form a proper subclass of the regular languages.

We extend regular expressions to *ω-regular expressions*.

- If e_1 and e_2 are regular expressions describing respectively the languages L_1 and L_2, then $e_1 e_2^\omega$ is an ω-regular expression that describes the ω-language $L_1 L_2^\omega$.

- If r_1 and r_2 are ω-regular expressions that describe respectively the ω-languages L_1 and L_2, then $r_1 + r_2$ and $\overline{r_1}$ are ω-regular expressions that describe respectively the ω-languages $L_1 \cup L_2$ and $\overline{L_1}$.

An *ω-regular language* is a language described by an ω-regular expression. ω-regular expressions built using star-free regular expressions are called *star-free ω-regular expressions*. These expressions define the so called *star-free ω-regular languages*, which form a proper subclass of ω-regular languages.

2.2.2 Finite Automata on Finite Words

Definition 2.2.1 (Finite Automata on Finite Words) *A finite automaton $\mathcal{A}$ over Σ is a tuple (S, s_0, δ, F) where S is a nonempty finite set of states, $s_0 \in S$ is the initial state, $\delta : S \times \Sigma \to \mathsf{Pow}(S)$ is the transition function, and $F \subseteq S$ is a set of final states. A run of $\mathcal{A}$ on a finite word $w_0,\ldots,w_{n-1}$ is a finite sequence of states $s_0, \ldots, s_n$ with $s_{i+1} \in \delta(s_i, a_i)$, for $0 \leq i < n$. A finite word is accepted by an automaton if it has a run whose last state is final. The language $L(\mathcal{A})$ denotes the set of the words accepted by $\mathcal{A}$. The* size *of $\mathcal{A}$ is defined as $|S|$. If $\delta(s, a)$ is either a singleton or empty for all $s \in S$ and $a \in \Sigma$, then $\mathcal{A}$ is called* deterministic finite automaton *(*DFA*) otherwise it is called* nondeterministic finite automaton *(*NFA*).*

Regular expressions, deterministic and nondeterministic finite automata on finite words are equiexpressive. That means any regular language can be accepted by a DFA and a NFA and, conversely, any language accepted by a DFA or an NFA can be defined by a regular expression. Like regular expressions, DFAs and NFAs are both closed under Boolean operations. In Table 2.1 we summarize some known complexity results for finite automata on finite words. The languages L_1 and L_2 are regular and the size of their DFAs are m and n respectively. The language L is arbitrary. In Table 2.2 we display the complexity of some problems for DFAs and NFAs. The automaton $\mathcal{A}$ has n states.

2.2.3 Finite Automata on Infinite Words

Definition 2.2.2 (Büchi Automata) *A Büchi automaton is a finite automaton on infinite words equipped with a so called* Büchi acceptance condition*, which says that an infinite word is accepted, if it has a run in which some final state occurs infinitely often. That is, for an infinite word $w = w_0, \ldots, w_{n-1}, \ldots$, there is an infinite sequence of states $s = s_0, \ldots, s_n, \ldots$ with $s_{i+1} \in \delta(s_i, a_i)$, for all $i \in \mathbb{N}$ and where some final state $s_i \in F$ occurs infinitely often in s. Analogously to finite automata on finite words, if $\delta(s, a)$ is either a singleton or empty for all $s \in S$ and*

operation	size of accepting automaton
$L_1 \cup L_2$	$m \cdot n$
$L_1 \cap L_2$	$m \cdot n$
L_1/L	m

Table 2.1: Closure operations and their complexity

problem	complexity
DFA	
emptiness $L(\mathcal{A}) = \emptyset$	*NLOGSPACE-complete*
equivalence $L(\mathcal{A}_1) = L(\mathcal{A}_2)$	*PSPACE-complete*
minimization of $\mathcal{A}$	$O(n \log n)$
NFA	
emptiness $L(\mathcal{A}) = \emptyset$	*NLOGSPACE-complete*
equivalence $L(\mathcal{A}_1) = L(\mathcal{A}_2)$	*PSPACE-complete*
minimization of $\mathcal{A}$	$O(n \log n)$

Table 2.2: DFA and NFA problems

$a \in \Sigma$, then $\mathcal{A}$ is called a deterministic Büchi automaton *(*DBA*) otherwise it is called a* nondeterministic Büchi automaton *(*NBA*).*

NBAs and ω-regular expressions are equiexpressive. The NBAs are closed under all Boolean operations, whereas the DBAs are closed under the Boolean operations except the negation. Thus, DBAs are strictly less expressive than NBAs.

2.2.4 Finite Automata on Finite Trees

Trees A (Σ-*labeled-*)*tree* t with branching factor $k \in \mathbb{N}$ over Σ is a function from D to Σ, where D is a prefix closed subset of $[k]^*$, *i.e.* (i) if $ui \in D$, then $u \in D$, and (ii) if $ui \in D$ then $uj \in D$, for all $j < i$. The elements of D are called *nodes* and the empty word $\epsilon \in D$ is called the *root*. The node $ui \in D$ is a *successor* of u. A node is an *inner node* if it has successors and is a *leaf* otherwise. If D is finite then t is called finite tree otherwise it is infinite. If D is empty then t is the empty tree that we denote with λ. We call t *k-ary* if every inner node $u \in D$ has exactly k successors. For $k = 2$, t is conventionally called a *binary* tree. We denote by T^k_Σ

(B_Σ, for $k = 2$) the set of all finite Σ-labeled k-ary trees.

For convenience and to simplify the technical parts of some proofs, we represent trees as terms. We write $f(t_0, \ldots, t_{k-1})$ for the tree whose root is labeled with f and that has the subtrees t_0,…, t_{k-1} at the positions 0, …, $k-1$, respectively. In a tree with branching factor k, we will identify a leaf labeled with a with the tree $a(t_0, \ldots, t_{k-1})$, where t_i is λ for $i < k$. We will suppress λ when writing trees as terms, for example, writing $f(a, b)$ for $f(a(\lambda, \lambda), b(\lambda, \lambda))$

Definition 2.2.3 (Bottom-up k-ary Tree Automata) *A bottom-up k-ary tree automaton $\mathcal{A}$ over Σ is tuple (S, S_0, δ, F), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $F \subseteq S$ is a set of final states and δ is a transition function $\delta : S^k \times \Sigma \to \mathsf{Pow}(S)$.*

A run r of a finite k-ary tree t is a finite k-ary S-labeled tree that is constructed in accordance to the transition function δ and the tree t in the usual way. Instead of formalizing the acceptance of trees using the notion of accepted runs, we use our definition which is more appropriate for our needs. We extend the transition function to trees $\widehat{\delta} : S^k \times T^k_\Sigma \to \mathsf{Pow}(S)$, such that $\widehat{\delta}(s_0, \ldots, s_{k-1}, t)$ yields all states that can label the roots of all runs of $\mathcal{A}$ on t starting from the states s_0, …, s_{k-1}. The function $\widehat{\delta}$ is given by: for $s \in S$, $\widehat{\delta}(s, \ldots, s, \lambda) = \{s\}$ and for $t = a(t_0, \ldots, t_{k-1})$,

$$\widehat{\delta}(s_0, \ldots, s_{k-1}, t) = \{s \in \delta(q_0, \ldots, q_{k-1}, a) \mid q_i \in \delta(s_0, \ldots, s_{k-1}, t_i),\ i < k\}\,.$$

Now, we say that a tree t is *accepted* by the automaton $\mathcal{A}$, if there is a $s \in F$ with $\widehat{\delta}(s_0, \ldots, s_{k-1}, t) \cap F \neq \emptyset$, where $s_i \in S_0$ for $i < k$. The *language accepted* by $\mathcal{A}$, denoted by $L(\mathcal{A})$, consists of all accepted trees. The class of languages accepted by bottom-up tree automata is called *the regular tree languages*.

If $\delta(s_0, \ldots, s_{k-1}, a)$ is either a singleton or empty for all $(s_0, \ldots, s_{k-1}) \in S$ and $a \in \Sigma$, then $\mathcal{A}$ is called *deterministic bottom-up k-ary tree automata* (DBTA) otherwise it is called *nondeterministic k-ary tree automata* (NBTA). Both DBTAs and NBTAs are closed under the Boolean operations and have the same expressiveness.

Example 2.2.1 Let $\Sigma = \{f, a, b\}$, $k = 2$, and the DBTA $\mathcal{A} = ([4], 0, \delta, \{3\})$, where the transition function δ is defined by

$$\begin{aligned} \delta(1, 2, f) &= \{3\}, & \delta(2, 1, f) &= \{3\}, \\ \delta(0, 0, a) &= \{1\}, & \delta(0, 0, b) &= \{2\}\,. \end{aligned}$$

We have $L(\mathcal{A}) = \{f(a, b), f(b, a)\}$. □

The bottom-up tree automata processes a tree from the leaves to the root. There is another kind of tree automata that process the trees in the opposite way, namely form the root to the leaves. This kind of automata is called *top-down tree automata* (cf, [GS84, Tha73]) and has the property that its deterministic version is less powerful as its nondeterministic version. It is known, for example, that the above language $L(\mathcal{A})$ is not acceptable by deterministic top-down binary tree automata. In the following we define a generalization of top-down tree automata.

Alternating Top-down Tree Automata

Alternating tree automata on words were introduced firstly in [BL80, CKS81] and on trees in [Slu85]. We use the definition of alternating automata on (infinite) words from [Var96] and adapt it to (finite) k-ary trees. For this purpose we need the following notions.

For a set X, let $B(X)$ be the set of Boolean formulae (including the truth values false and true) over X, built using the connectives $\wedge$, $\vee$ and $\neg$. We use $\oplus$ as a symbol that stands for either $\wedge$ or $\vee$. For $S \subseteq X$ and $e \in B(X)$, we write $S \models e$, if S satisfies e, that is, if the truth assignment that assigns true to the variables in S and false to the variables in $X \setminus S$ satisfies e, we say that S is a *model* of e. We define $Mod(e)$ as the set of all models of e. Below we will instantiate X with both the set S of states and the set $S \times [k]$, with $k \in \mathbb{N}$. We call the elements of $B(S)$ unary state expressions and the elements of $B(S \times [k])$, with $k > 1$, k-ary state expressions. We write q^i for $(q, i) \in S \times [k]$.

Let Ξ be the function $\Xi : B(S \times [k]) \to B(S)$ defined by

$$\Xi(b) = \begin{cases} q, & \text{if } b = q^i \\ \neg\Xi(b_1), & \text{if } b = \neg b_1 \\ \Xi(b_1) \oplus \Xi(b_2), & \text{if } b = b_1 \oplus b_2 \end{cases}$$

Intuitively, the function Ξ converts a k-ary state expression into an unary state expression by deleting i (the successor position) form any k-ary state q^i. For Ξ the following fact holds.

Fact 2.2.4 *For each k-ary state expression $b \in B(S \times [k])$ and each $E \subseteq S$,*

$$E \models \Xi(b) \textit{ iff } (E \times [k]) \models b$$

Fact 2.2.4 follows by induction over the structure of k-ary state expressions.

Definition 2.2.5 (Alternating Top-down Tree Automata (ATTA)) *An* alternating top-down tree automaton $\mathcal{A}$ *over* Σ *is a tuple* (S, I, δ, F)*, where* S *is a set of states,* $I \in B(S)$ *is an initial Boolean formula,* $F \subseteq S$ *is set of final states and* δ *is a transition function* $\delta : S \times \Sigma \to B(S \times [k])$.

In the same way as for bottom-up tree automata, we extend the transition function δ to unary state expressions and trees: $\widehat{\delta} : B(S) \times T^k_\Sigma \to B(S)$. The function $\widehat{\delta}$ uses a help function $\check{\delta} : B(S \times [k]) \times (T^k_\Sigma)^k \to B(S)$. The two functions are defined as follows:

$$\widehat{\delta}(u,t) = \begin{cases} u, & \text{if } u \in \{\mathsf{true}, \mathsf{false}\} \text{ or } t = \lambda \\ \neg\widehat{\delta}(u_1,t), & \text{if } u = \neg u_1 \\ \widehat{\delta}(u_1,t) \oplus \widehat{\delta}(u_2,t), & \text{if } u = u_1 \oplus u_2 \\ \check{\delta}(\delta(q,a), t_0, \ldots, t_{k-1}), & \text{if } u = q \text{ and } q \in S \text{ and } t = a(t_0, \ldots, t_{k-1}) . \end{cases}$$

and

$$\check{\delta}(u, t_0, \ldots, t_{k-1}) = \begin{cases} u, & \text{if } u \in \{\mathsf{true}, \mathsf{false}\} \\ \neg\check{\delta}(u', t_0, \ldots, t_{k-1}), & \text{if } u = \neg u' \\ \check{\delta}(u_1, t_0, \ldots, t_{k-1}) \oplus \check{\delta}(u_2, t_0, \ldots, t_{k-1}) & \text{if } u = u_1 \oplus u_2 \\ \widehat{\delta}(q, t_i), & \text{if } u = q^i \in S \times [k] \end{cases}$$

The function $\widehat{\delta}$ applied to a state q and a tree t computes a state expression whose disjunctive normalform is a formula $u_1 \vee \ldots \vee u_n$, where the states occurring in a conjunction u_i constitute the leaves of a run of t starting from the state q. For example, if a tree t has in $\mathcal{A}$ starting from a state q only two runs r with leaves $\{q_1, q_2\}$ and s with leaves $\{s_1, s_2, s_3\}$, then $\widehat{\delta}(q,t) = (q_1 \wedge q_2) \ \vee \ (s_1 \wedge s_2 \wedge s_3)$.

We say that an automaton $\mathcal{A} = (S, I, \delta, F)$ *accepts* a tree t, if $F \models \widehat{\delta}(I,t)$ holds. The language accepted by $\mathcal{A}$ is defined by $L(\mathcal{A}) = \{t \mid F \models \widehat{\delta}(I,t)\}$.

Example 2.2.2 We give an alternating top-down binary tree automaton that accepts the same language as the bottom-up binary tree automaton given in Example 2.2.1. The automaton is $\mathcal{A} = (\{q_0, q_1, q_2, q_3\}, q_0, \delta, \{q_3\})$, where the transition function δ is defined by:

$$\delta(q,x) = \begin{cases} q_1^0 \wedge q_2^1 \ \vee \ q_2^0 \wedge q_1^1, & \text{if } q = q_0 \text{ and } x = f \\ q_3^0 \wedge q_3^1, & \text{if } q = q_1 \text{ and } x = a \\ q_3^0 \wedge q_3^1, & \text{if } q = q_2 \text{ and } x = b \\ \mathsf{false}, & \text{otherwise.} \end{cases}$$

We show that $f(a,b) \in L(\mathcal{A})$.

$$\begin{aligned}
\widehat{\delta}(q_0, f(a,b)) &= \check{\delta}(\delta(q_0,f),a,b) \\
&= \check{\delta}((q_1^0 \wedge q_2^1) \vee (q_2^0 \wedge q_1^1), a, b) \\
&= \check{\delta}(q_1^0,a,b) \wedge \check{\delta}(q_2^1,a,b) \vee \check{\delta}(q_2^0,a,b)) \wedge \check{\delta}(q_1^1,a,b) \\
&= \widehat{\delta}(q_1,a) \wedge \widehat{\delta}(q_2,b) \vee \widehat{\delta}(q_2,a) \wedge \widehat{\delta}(q_1,b) \\
&= \check{\delta}(q_3^0,\lambda,\lambda) \wedge \check{\delta}(q_3^1,\lambda,\lambda) \wedge \check{\delta}(q_3^0,\lambda,\lambda) \wedge \check{\delta}(q_3^1,\lambda,\lambda) \vee \mathsf{false} \wedge \mathsf{false} \\
&= q_3 \wedge q_3 \wedge q_3 \wedge q_3 \vee \mathsf{false} \wedge \mathsf{false} \\
&= q_3
\end{aligned}$$

Since $\{q_3\} \models q_3$ thus $f(a,b) \in L(\mathcal{A})$. Similarly, we can show that $f(b,a) \in L(\mathcal{A})$. Now, we show for instance that $f(a,a) \notin L(\mathcal{A})$.

$$\begin{aligned}
\widehat{\delta}(q_0, f(a,a)) &= \check{\delta}(\delta(q_0,f),f(a,a)) \\
&= \check{\delta}(q_1^0 \wedge q_2^1 \vee q_2^0 \wedge q_1^1, a, a) \\
&= \check{\delta}(q_1^0,a,a) \wedge \check{\delta}(q_2^1,a,a) \vee \check{\delta}(q_2^0,a,a) \wedge \check{\delta}(q_1^1,a,a) \\
&= \widehat{\delta}(q_1,a) \wedge \widehat{\delta}(q_2,a) \vee \widehat{\delta}(q_2,a) \wedge \widehat{\delta}(q_1,a) \\
&= q_3 \wedge q_3 \wedge \mathsf{false} \vee \mathsf{false} \wedge q_3 \wedge q_3 \\
&= \mathsf{false}
\end{aligned}$$

The claim holds, because $\{q_3\} \not\models \mathsf{false}$. □

Specialization of Alternating Top-down Tree Automata

We notice that in Definition 2.2.5 we used a Boolean formula I to encode the initial states instead of using a state or set of states as it is the case in the definition given in [Var96]. Our choice to use an initial Boolean formula is useful for some constructions and helps keeping proofs simple.

If we take k to be the natural number 1 in Definition 2.2.5, we obtain a definition of *alternating word automata* (AWA), which slight differs from the definition given in [Var96] on acceptance condition. In our definition, the acceptance of words is defined using an extension of the transition function δ, whereas in [Var96] it is defined in terms of accepted runs. We can easily prove that our definition of AWA and the one in [Var96] are equivalent.

Now, if we take k to be the natural number 2, require that the transition function δ from Definition 2.2.5 has the general form $\delta(q,a) = (l_1^0 \wedge r_1^1) \vee \ldots \vee (l_m^0 \wedge r_m^1)$, where $l_i, r_i \in S$ for $i \leq m$, and interpret the conjunction $l_i^0 \wedge r_i^0$ by the pair (l_i, r_i), then we obtain an equivalent definition of the conventional *nondeterministic top-down binary tree automata.*

We show that alternation does not increase the expressiveness of word and tree automata but, as we will see, it does enhance their ability to model problems. For the sake of simplicity, we will consider only binary trees. The results can be straightforwardly generalized for k-ary trees for $k \geq 1$. In the following, we will use the pair (l, r) and the formula $l^0 \wedge r^1$ interchangeably.

Theorem 2.2.6 *For every* ATTA $\mathcal{A} = (S, I, \delta, F)$ *with n states there is a* DBTA $\mathcal{B}$ *with at most 2^n states, such that $L(\mathcal{A}) = L(\mathcal{B})$. The automaton $\mathcal{B}$ is given by* $\mathcal{B} = (\mathsf{Pow}(S), \{F\}, \gamma, \mathit{Mod}(I))$, *where* $\gamma\colon \mathsf{Pow}(S) \times \mathsf{Pow}(S) \times \Sigma \to \mathsf{Pow}(S)$,

$$\gamma(E_0, E_1, a) = \{q \mid M \models \delta(q, a)\},\ \textit{where } M = \{q^0 \mid q \in E_0\} \cup \{q^1 \mid q \in E_1\}\,.$$

We extend γ to trees as follows: $\widehat{\gamma}\colon \mathsf{Pow}(S) \times \mathsf{Pow}(S) \times B_\Sigma \to \mathsf{Pow}(S)$

$$\widehat{\gamma}(E_0, E_1, t) = \begin{cases} E_0, & \textit{if } E_0 = E_1 \textit{ and } t = \lambda \\ \gamma(\widehat{\gamma}(E_0, E_1, t_0), \widehat{\gamma}(E_0, E_1, t_1), a), & \textit{if } t = a(t_0, t_1) \end{cases}$$

In order to prove Theorem 2.2.6, we first establish the following properties.

Lemma 2.2.7 *Let $\mathcal{A}$ and $\mathcal{B}$ be automata as described in Theorem 2.2.6. For t_1, $t_2 \in B_\Sigma$, $b \in B(S \times [2])$ and $E \subseteq S$, it holds:*

$$E \models \check{\delta}(b, t_1, t_2) \textit{ iff } \{q^0 \mid q \in \widehat{\gamma}(E, E, t_1)\} \cup \{q^1 \mid q \in \widehat{\gamma}(E, E, t_2)\} \models b\,.$$

Proof By simultaneous induction over the structure of the trees t_1 and t_2. If $t_1 = \lambda$ and $t_2 = \lambda$, then we can show that $\check{\delta}(b, \lambda, \lambda) = \Xi(b)$ and $\widehat{\gamma}(E, E, \lambda) = E$. In this case the goal can be reduced to $E \models \Xi(b)$ iff $E \times [2] \models b$, which holds by Fact 2.2.4. The cases where one of the trees t_1 or t_2 is empty are analogous to the case where both trees aren't empty. So, suppose $t_1 = a^1(t_1^1, t_2^1)$ and $t_2 = a^2(t_1^2, t_2^2)$. Our goal is to prove that for each $b \in B(S \times [2])$ and $E \subseteq S$, it holds:

$$E \models \check{\delta}(b, t_1, t_2) \text{ iff } \{q^0 \mid q \in \widehat{\gamma}(E, E, t_1)\} \cup \{q^1 \mid q \in \widehat{\gamma}(E, E, t_2)\} \models b\,.$$

and the induction hypothesis is: for each $b \in B(S \times [2])$, $E \subseteq S$, and $i = 1, 2$

$$E \models \check{\delta}(b, t_1^i, t_2^i) \ \text{ iff } \{q^0 \mid q \in \widehat{\gamma}(E, E, t_1^i)\} \cup \{q^1 \mid q \in \widehat{\gamma}(E, E, t_2^i)\} \models b\,. \tag{2.1}$$

We proceed by induction over b. The cases where $b = \mathsf{true}$, $b = \mathsf{false}$, $b = b_1 \oplus b_2$ and $b = \neg b_1$ are trivial. Let us consider the case where $b = q^i$, with $i = 0, 1$. By using the definitions of $\widehat{\delta}$, $\check{\delta}$, and $\widehat{\gamma}$, the goal can be transformed into

$$E \models \check{\delta}(\delta(q_1, a^i), t_1^i, t_2^i) \text{ iff } \{q^0 \mid q \in \widehat{\gamma}(E, E, t_1^i)\} \cup \{q^1 \mid q \in \widehat{\gamma}(E, E, t_2^i\} \models \delta(q_1, a^i)\,,$$

which follows immediately from the induction hypothesis (2.1). ■

Lemma 2.2.8 *Given $\mathcal{A}$ and $\mathcal{B}$ automata as in Theorem 2.2.6. For $t \in B_\Sigma$, $q \in S$ and the final set F of $\mathcal{A}$ (the single initial state of $\mathcal{B}$), it holds: $F \models \widehat{\delta}(q,t)$ iff $q \in \widehat{\gamma}(F,F,t)$.*

Proof By induction over t. If $t = \lambda$, then by definition $\widehat{\delta}(q,\lambda) = q$ and $\widehat{\gamma}(F,F,\lambda) = F$. In this case our goal can be reduced to the trivial fact $F \models q$ iff $q \in F$. Next, suppose $t = a(t_1,t_2)$. We instantiate b and E in Lemma 2.2.7 with $\delta(q,a)$ and F respectively:

$$F \models \check{\delta}(\delta(q,a),t_1,t_2) \text{ iff } \{q^0 \mid q \in \widehat{\gamma}(F,F,t_1)\} \cup \{q^1 \mid q \in \widehat{\gamma}(F,F,t_2\} \models \delta(q,a)$$

By unfolding the definitions of $\widehat{\delta}$ and $\widehat{\gamma}$, we can simplify our goal to

$$F \models \widehat{\delta}(q,a(t_1,t_2)) \text{ iff } q \in \gamma(\widehat{\gamma}(F,F,t_1),\widehat{\gamma}(F,F,t_2),a)$$

and this by the definition of γ establishes our main goal. ■

Using the above lemmata we can prove Theorem 2.2.6.

Proof (of Theorem 2.2.6) By definition, $t \in L(\mathcal{A})$ iff $F \models \widehat{\delta}(I,t)$ and $t \in L(\mathcal{B})$ iff $\widehat{\gamma}(F,F,t) \models I$. Further, by Lemma 2.2.8, $F \models \widehat{\delta}(I,t)$ iff $\widehat{\gamma}(F,F,t) \models I$ holds and thus $t \in L(\mathcal{A})$ iff $t \in L(\mathcal{B})$. ■

Example 2.2.3 Following Theorem 2.2.6, the automaton $\mathcal{B}$ over $\Sigma = \{f,a,b\}$ constructed from $\mathcal{A}$ of Example 2.2.2 is given by $\mathcal{B} = (\mathsf{Pow}(S), S_0, \gamma, F)$, where $S = \{q_0,q_1,q_2,q_3\}$, $S_0 = \{\{q_3\}\}$, $F = Mod(q_0) = \{Q \subseteq S \mid q_0 \in Q\}$), the transition function γ is defined as follows: $\gamma(\{q_1\},\{q_2\},f) = \gamma(\{q_2\},\{q_1\},f) = \{q_0\}$, $\gamma(\{q_3\},\{q_3\},a) = \{q_1\}$ and $\gamma(\{q_3\},\{q_3\},b) = \{q_2\}$. The automaton $\mathcal{B}$ is, up to state renaming, isomorphic to the automaton $\mathcal{A}$ of Example 2.2.1. □

We can easily prove that ATTA are closed under the Boolean operations. For $k = 1$, Theorem 2.2.6 needs some explanation. So for $k = 1$, ATTA are just AWA and the constructed automaton $\mathcal{B}$ is a DFA that accepts the reverse language that is accepted by $\mathcal{A}$. Note that in this case $\mathcal{B}$ can be further transformed into a DFA, say $\mathcal{C}$, that accepts the same language as $\mathcal{A}$. In the worst case the number of states of $\mathcal{C}$ is exponential in the number of states of $\mathcal{B}$ and thus double exponential in the number of states of $\mathcal{A}$. It is shown in [BL80, CKS81, Lei81b] that this double exponential blow-up is unavoidable.

We extend now the transition rules of ATTAs with a new type of rule, ϵ-rule (ϵ-transition), that allows the move from a state to a successor state without reading any input symbol.

Definition 2.2.9 (Alternating Top-down Tree Automata with ϵ-transitions) *An* ATTA *with ϵ-transitions over Σ is a tuple $\mathcal{A} = (S, q_0, \delta, \delta_\epsilon, F)$, where (S, q_0, δ, F) is an* ATTA *and δ_ϵ is a function (also called ϵ-transition function) $\delta_\epsilon : S \to B(S)$.*

Let $\Delta \subseteq B(S) \times T_\Sigma^k \times B(S)$ be the graph of the function $\widehat{\delta}$, which extends δ as described above. Define the ϵ-closure of Δ, as the relation Δ^*, to be the smallest relation which includes Δ and satisfies the following rule: If $(u, t, v) \in \Delta^*$ and $w \in B(S)$ is constructed from v by replacing some occurrences of a state q in v with $\delta_\epsilon(q)$, then $(u, t, w) \in \Delta^*$. We say that $\mathcal{A}$ accepts a tree t, if there is an unary Boolean state $u \in B(S)$ such that $(q_0, t, u) \in \Delta^*$ and $F \models u$.

As it is known for word automata, ϵ-transitions may be useful but really add no more power. In the subsequent sections we describe how we eliminate the ϵ-transitions and report on the required computation overhead.

Let $S = \{q_1, \ldots, q_n\}$, and C_ϵ be the endomorphism on $B(S)$, which associates u with

$$C_\epsilon(u) = u[(q_1 \vee \delta_\epsilon(q_1))/q_1, \ldots, (q_n \vee \delta_\epsilon(q_n))/q_n]\,.$$

Intuitively, C_ϵ computes from u a new state expression where all states q are simultaneously replaced with the disjunction $q \vee \delta_\epsilon(q)$. The expressions u and $C_\epsilon(u)$ are logically equivalent, if we assume that q and $\delta_\epsilon(q)$ are logically equivalent.

We extend C_ϵ homomorphically to k-ary state expressions.

$$C_\epsilon(b) = \begin{cases} (C_\epsilon(q))[q_1/q_1^i, \ldots, q_n/q_n^i], & \text{if } b = q^i \in S \times [k] \\ C_\epsilon(b_1) \oplus C_\epsilon(b_2), & \text{if } b = b_1 \oplus b_2 \\ \neg C_\epsilon(b_1), & \text{if } b = \neg b_1 \end{cases}$$

Furthermore, we define the iteration of the function C_ϵ in the usual way:

$$\begin{array}{lcl} C_\epsilon^0(u) & = & u \\ C_\epsilon^{n+1}(u) & = & C_\epsilon(C_\epsilon^n(u)) \end{array}$$

In order to eliminate the ϵ-transition of a state q, we have to compute all state expressions reachable by successively iterating C_ϵ on q. Because there are at most 2^{2^n} Boolean functions over S, the number of iterations needed for C_ϵ is then at most 2^{2^n}. With this at hand, define the ϵ-closure C_ϵ^* to be the function $C_\epsilon^*(u) = C_\epsilon^{2^{2^n}}(u)$.

Theorem 2.2.10 *For every* ATTA *$\mathcal{A}$ with ϵ-transitions there is an* ATTA *(without ϵ-transitions) $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$. Furthermore, $\mathcal{A}$ and $\mathcal{B}$ have the same number of states.*

Proof Let $\mathcal{A} = (S, q_0, \delta, \delta_\epsilon, F)$. The automaton $\mathcal{B}$ is constructed as follows: $\mathcal{B} = (S, q_0, \delta', F')$, where

$$F' = \begin{cases} F \cup \{q_0\}, & \text{if } F \models C^*_\epsilon(q_0) \\ F, & \text{otherwise.} \end{cases}$$

and the transition function $\delta' : S \times \Sigma \to B(S \times [k])$ is given by $\delta'(q, a) = C^*_\epsilon(\delta(C^*_\epsilon(q), a))$. By the construction of $\mathcal{B}$, it follows that $L(\mathcal{A}) = L(\mathcal{B})$. ■

In the rest of this chapter, we introduce the logics used in this thesis. We start with quantified Boolean logic.

2.3 Quantified Boolean Logic

Boolean formulae are built from the constants true and false, variables $x \in \mathcal{V}$, and are closed under the standard connectives. The formulae are interpreted in $\mathbb{B} = \{0, 1\}$. A (Boolean) *substitution* $\sigma : \mathcal{V} \to \mathbb{B}$ is a mapping from variables to truth values that is extended homomorphically to formulae. We say σ *satisfies* ϕ if $\sigma(\phi) = 1$.

Quantified Boolean logic (QBL) extends Boolean logic (BL) by allowing quantification over Boolean variables, *i.e.* $\forall x.\, \phi$ and $\exists x.\, \phi$. A substitution σ satisfies $\forall x.\, \phi$ if σ satisfies $\phi[\mathsf{true}/x] \wedge \phi[\mathsf{false}/x]$ and dually for $\exists x.\, \phi$. In the remainder of the thesis, we write $\sigma \models_{\text{QBL}} \phi$ to denote that σ satisfies ϕ in QBL.

QBL is not more expressive than BL, but it is more succinct. The satisfiability problem for Boolean logic is *NP-complete* [Coo71], whereas it is *PSPACE-complete* for QBL [MS73].

2.4 Monadic Second-Order Logics on Words

The monadic logics M2L-Str, WS1S, and S1S are among the most expressive decidable logics known. The logic M2L-Str [HJJ+96] is a logic on finite words and also appears in the literature (with slight variations) under the names MSO[S] [Tho90] and SOM[+1] [Str94]. WS1S stands for the *Weak Second-order Logic of One Successor* and S1S stands for the *Second-order Logic of One Successor*.

In the early 1960's, Büchi and Elgot gave decision procedures for these logics by exploiting the fact that models can be encoded as words and that the language of models satisfying a formula can be represented by an automaton [Büc60, Büc62, Elg61]. These decision procedures provide non-elementary upper-bounds for these logics, which are also the lower-bounds [Mey75].

In this section we provide background material on M2L-STR, WS1S and S1S. These logics have the same syntax but slightly different semantics. We also explain their relationship to regular languages.

Let $\mathcal{V}_1 = \{x_i \mid i \in \mathbb{N}\}$ be a set of first-order variables and $\mathcal{V}_2 = \{X_i \mid i \in \mathbb{N}\}$ be a set of second-order variables. We will use n, p, q, ... as meta-variables ranging over $\mathcal{V}_1$ and we use X, Y, ... as meta-variables ranging over $\mathcal{V}_2$. We will often use the alphabet $\mathbb{B}^n$, with $n \in \mathbb{N}$. Note that $\mathbb{B}^0$ stands for the singleton set $\{()\}$, *i.e.* the set whose only member is the degenerate tuple "()".

2.4.1 Language

Monadic second-order (MSO) formulae are formulae in a language of second-order arithmetic specified by the grammar:

$$t ::= \mathsf{0} \mid p, \qquad p \in \mathcal{V}_1$$

$$\phi ::= \mathsf{s}(t,t) \mid X(t) \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\, \phi \mid \exists X.\, \phi, \qquad p \in \mathcal{V}_1 \text{ and } X \in \mathcal{V}_2$$

Hence terms are built from the constant $\mathsf{0}$ and first-order variables. Formulae are built from predicates $\mathsf{s}(t,t')$ and $X(t)$ and are closed under disjunction, negation, and quantification over first-order and second-order variables. Other connectives and quantifiers can be defined using standard classical equivalences, *e.g.* $\forall X.\, \phi \overset{def}{=} \neg\exists X.\, \neg\phi$. In other presentations, s is usually a function. We have specified it as a relation for reasons that will become apparent when we give the semantics. In the remainder of this section, formula means MSO-formula.

2.4.2 Semantics

A (MSO) substitution σ is a pair of mappings $\sigma = (\sigma_1, \sigma_2)$, with $\sigma_1{:}\mathcal{V}_1 \to \mathbb{N}$ and $\sigma_2{:}\mathcal{V}_2 \to \mathsf{Pow}(\mathbb{N})$ and for $x \in \mathcal{V}_1$, $\sigma(x) = \sigma_1(x)$ and for $X \in \mathcal{V}_2$, $\sigma(X) = \sigma_2(X)$. With this at hand, we can now define satisfiability for M2L-STR and WS1S.

2.4.3 The Logic M2L-Str

Formulae in M2L-STR are interpreted relative to a natural number k. We call the elements of $[k]$ *positions*. First-order variables are interpreted as positions. The constant $\mathsf{0}$ denotes the natural number 0 and the symbol s is interpreted as the relation $\{(i,j) \mid j = i+1 \text{ and } i,j \in [k]\}$. Note that $k-1$ has no successor. Second-order variables denote subsets of $[k]$ and the formula $X(t)$ is true when the position denoted by t is in the set denoted by X.

More formally, the semantics of a formula ϕ is defined inductively relative to a substitution σ and a $k \in \mathbb{N}$. In the following, we write σ^k for the pair (σ, k).

Definition 2.4.1 *Satisfiability for* M2L-STR

$$\begin{array}{lll}
\sigma^k \models_{M2L} \mathsf{s}(t,t'), & \textit{if} & \sigma(t') = 1 + \sigma(t) \textit{ and } \sigma(t') \in [k] \\
\sigma^k \models_{M2L} X(t), & \textit{if} & \sigma(t) \in \sigma(X) \\
\sigma^k \models_{M2L} \neg\phi, & \textit{if} & \sigma^k \not\models_{M2L} \phi \\
\sigma^k \models_{M2L} \phi_1 \vee \phi_2, & \textit{if} & \sigma^k \models_{M2L} \phi_1 \textit{ or } \sigma^k \models_{M2L} \phi_2 \\
\sigma^k \models_{M2L} \exists p.\ \phi, & \textit{if} & (\sigma[i/p])^k \models_{M2L} \phi, \textit{ for some } i \in [k] \\
\sigma^k \models_{M2L} \exists X.\ \phi, & \textit{if} & (\sigma[M/X])^k \models_{M2L} \phi, \textit{ for some } M \subseteq [k]
\end{array}$$

If $\sigma^k \models_{\text{M2L}} \phi$, we say that σ^k *satisfies*, or is a *model* of, ϕ. We call a formula ϕ *valid*, and we write $\models_{\text{M2L}} \phi$, if for every natural number k and substitution σ, σ^k satisfies ϕ.

2.4.4 The Logic WS1S

Whereas M2L-STR can be seen as a logic on bounded sets of positions or, as we shall see, finite words, WS1S is best viewed as a logic based on arithmetic. First-order variables range over $\mathbb{N}$ and are not a priori bounded by any natural number. Second-order variables range over finite subsets of the natural numbers, $\mathsf{Pow}(\mathbb{N})$, and are not restricted to subsets of some $[k]$. Finally, the symbol s is interpreted as the successor relation over $\mathbb{N}$. Formally, we define satisfiability in WS1S, $\sigma \models_{\text{WS1S}} \phi$, as follows:

Definition 2.4.2 *Satisfiability for* WS1S

$$\begin{array}{lll}
\sigma \models_{WS1S} \mathsf{s}(t,t'), & \textit{if} & \sigma(t') = 1 + \sigma(t) \\
\sigma \models_{WS1S} X(t), & \textit{if} & \sigma(t) \in \sigma(X) \\
\sigma \models_{WS1S} \neg\phi, & \textit{if} & \sigma \not\models_{WS1S} \phi \\
\sigma \models_{WS1S} \phi_1 \vee \phi_2, & \textit{if} & \sigma \models_{WS1S} \phi_1 \textit{ or } \sigma \models_{WS1S} \phi_2 \\
\sigma \models_{WS1S} \exists p.\ \phi, & \textit{if} & \sigma[i/p] \models_{WS1S} \phi, \textit{ for some } i \in \mathbb{N} \\
\sigma \models_{WS1S} \exists X.\ \phi, & \textit{if} & \sigma[M/X] \models_{WS1S} \phi, \textit{ for some finite subset } M \in \mathsf{Pow}(\mathbb{N})
\end{array}$$

A formula ϕ is *valid* in WS1S (we write $\models_{\text{WS1S}} \phi$) if it is satisfied by every substitution σ.

Additional Syntax

As we mentioned before the standard connectives and quantifiers that are not part of the syntax of MSO formulae can be defined as expected. We define here additional syntax. We will write $t+1$ for t' for which $\mathsf{s}(t,t')$ holds and write $t-1$ for t' for which $\mathsf{s}(t',t)$ holds. Note that in M2L-STR the terms $t+1$ and $t-1$ do not always exists because the relation s is partial. The equality relation between positions and sets can be defined as follows:

$$t = t' \stackrel{def}{=} \forall X.\, X(t) \Leftrightarrow X(t'), \text{ and}$$

$$X = Y \stackrel{def}{=} \forall p.\, X(p) \Leftrightarrow Y(p)\,.$$

We use the constant $\emptyset$ to denote the empty set, the set X that satisfies $\forall p.\, \neg X(p)$. The set operations $X \cup Y$, $X \cap Y$, and $\overline{X}$ can be also defined in the appropriate way. For example, $X \cup Y$ can be defined as the set Z that satisfies $\forall p.\, (X(p) \vee Y(p)) \Leftrightarrow Z(p)$. The less-than relation $<$ is definable in M2L-Str and WS1S as follows.

$$t < t' \stackrel{def}{=} \forall X.\, (X(t') \wedge (\forall p.\, X(p) \rightarrow X(p-1))) \rightarrow X(t+1)$$

We will use also the above definitions in S1S (see Section 2.4.6). In M2L-Str, we use the constant \$[1] to denote the natural number k of Definition 2.4.1. Intuitively, if we identify substitutions and words (as we will do later), then \$ denotes the last position in a word. For instance, the formula $\$ < 10$ describes all words of length less than 10.

Finally, Boolean connectives and Booleans as well as quantification over Booleans are not part of MSO syntax, but can be straightforwardly encoded. For a Boolean b, we associate a second-order Variable B and we encode occurrences of b in a formula by the (MSO) formula $B(0)$. In this way, Boolean quantification is encoded using second-order quantification. For example, the Boolean formula $\forall a, b.\, a \wedge b$ is encoded by the (MSO) formula $\forall A, B.\, A(0) \wedge B(0)$.

Now, we have three types of variables in MSO: first-order, second-order, and Boolean variables. For the sake of clarity and to help disambiguation, we will use capital letters for second-order variables, lower-case letters like p, q, i, j, x for first-order variables, and also lower-case letters like a, b, c for Boolean variables. Generally, the order of a variable can be inferred from the context where it appears.

Word Models

Models in both M2L-Str and WS1S can be encoded as finite words. Let $\phi(\overline{X})$ be a formula, where $\overline{X}$ is the tuple of second-order variables $X_1, \ldots, X_n$ occurring free in ϕ.[2] We encode a M2L-Str model σ^k for ϕ by the word $w_{\sigma^k} \in (\mathbb{B}^n)^k$, such that the length of w_{σ^k} is k and for every position $i \in [k]$, $w_{\sigma^k}(i) = (b_1, \ldots, b_n)$ and for $1 \leq j \leq n$, $b_j = 1$ iff $i \in \sigma(X_j)$. We call w_{σ^k} *a word model* for ϕ and define $\mathcal{L}_{\text{M2L}}(\phi)$ as the set of all M2L-Str word models for ϕ. We shall also write $w \models_{\text{M2L}} \phi$ for $\sigma^k \models_{\text{M2L}} \phi$, if w encodes σ^k.

[1]As a syntactic sugar

[2]First-order variables can be encoded using second-order variables as we will show later.

Similarly, a WS1S model σ for ϕ can be encoded as a finite word w_σ such that $w_\sigma(i) = (b_1, \ldots, b_n)$ where $b_j = 1$ iff $i \in \sigma(X_j)$. We also call w_σ a *word model* for ϕ in WS1S. We define $\mathcal{L}_{\text{WS1S}}(\phi)$ as the set of WS1S word models for ϕ. Note that the encoding of M2L-STR models as words is a bijection, whereas this is not the case for WS1S. In particular, if σ is a WS1S model and w_σ encodes it, then any finite word of the form $w_\sigma aa \cdots a$, where a is $(0, \ldots, 0) \in \mathbb{B}^n$, also encodes σ. We shall also write $w \models_{\text{WS1S}} \phi$ for $\sigma \models_{\text{WS1S}} \phi$, where w encodes σ.

Example

Consider the formula $\phi \overset{def}{=} X(0) \wedge \forall p. X(p) \leftrightarrow Y(p+1)$ and the substitution σ with $\sigma(X) = \{0, 2\}$ and $\sigma(Y) = \{0, 1, 3\}$. σ^4 is a model for ϕ in M2L-STR and σ is a model for ϕ in WS1S. The words w and w' below encode σ^4 and σ, respectively.

w	0	1	2	3
X	1	0	1	0
Y	1	1	0	1

w'	0	1	2	3	4	5
X	1	0	1	0	0	0
Y	1	1	0	1	0	0

As a second example, the formula $\exists X. \forall p. X(p)$ is valid in M2L-STR, whereas it is unsatisfiable in WS1S.

Decidability

We have seen that monadic formulae define sets of word models. Büchi and Elgot in [Büc60, Elg61] and independently Trakhtenbrot in [TB73] proved that the languages formalized by formulae in WS1S and M2L-STR are regular and, conversely, that every regular language is both WS1S and M2L-STR-definable. To show regularity, they proved constructively that, given a formula ϕ, there exists an automaton A_ϕ that accepts all WS1S (respectively M2L-STR) word models for ϕ. This construction yields a decision procedure: a closed formula is valid in WS1S (respectively in M2L-STR) iff its corresponding automaton accepts the language $()^*$. This decision procedure (and indeed any decision procedure for these logics) is non-elementary [Mey75, Tho97]: the minimal automaton representing a formula of size n may require space whose lower bound is a stack of exponentials of height n.

The compilations of monadic formulae into finite automata are known and can be found in many text books about finite automata and monadic logics. We want, however, to sketch the compilation of M2L-STR and WS1S formulae into finite automata on finite words to essentially emphasis the difference between M2L-STR and WS1S and to anticipate a comparison of these logics in Chapter 4.

To simplify matters, we reduce MSO to a minimal kernel, called MSO$_0$, which is as expressive as MSO. The language MSO_0 has the grammar:

$$\phi ::= \mathsf{Succ}(X,Y) \mid X \subseteq Y \mid \neg\phi \mid \phi \vee \phi \mid \exists X.\ \phi, \qquad X,Y \in \mathcal{V}_2\ .$$

$\mathsf{Succ}(X,Y)$ means that X and Y are singletons $\{p\}$ and $\{q\}$, where $q = p+1$. The symbol $\subseteq$ denotes the subset relation. Note that first-order variables are omitted as they can be encoded as singletons. There is a simple polynomial time translation $\lceil . \rceil$ from MSO formulae into MSO_0 [Tho90]. We omit here a formal definition of the semantics of MSO_0.

In the following, we use the shorthand $P = Q$ for $P \subseteq Q \wedge Q \subseteq P$. Now, we define the predicates $\mathsf{Empty}(P)$ and $\mathsf{Singleton}(P)$ where the first states that the set P is empty and the second states that the set P is a singleton.

$$\begin{aligned} \mathsf{Empty}(P) &= \forall Y.\, P \subseteq Y \\ \mathsf{Singleton}(P) &= \forall Y.\, Y \subseteq P \wedge \neg\mathsf{Empty}(Y) \Rightarrow P \subseteq Y \end{aligned}$$

The translation $\lceil . \rceil$ is given by the following rules.

$$\begin{aligned} \lceil \mathsf{s}(x,y) \rceil &= \mathsf{Succ}(X,Y) \\ \lceil Y(x) \rceil &= \mathsf{Singleton}(X) \wedge X \subseteq Y \\ \lceil \phi(\mathsf{0}) \rceil &= \exists P.\, \mathsf{Singleton}(P) \wedge (\neg\exists Q.\, \mathsf{Succ}(P,Q)) \wedge \lceil \phi(P) \rceil, \ P \text{ and } Q \text{ are fresh} \\ \lceil \exists p.\, \phi \rceil &= \exists P.\, \mathsf{Singleton}(P) \wedge \lceil \phi \rceil \\ \lceil \exists X.\, \phi \rceil &= \exists X. \lceil \phi \rceil \end{aligned}$$

As mentioned before, due to Büchi, Elgot, and Trakhtenbrot we have the following result.

Theorem 2.4.3 *Let ϕ be an MSO_0 formula with n free variables. Then it holds*

(i) There is an automaton $\mathcal{A}_{WS1S}(\phi)$ over $\mathbb{B}^n$ such that $\mathcal{L}_{WS1S}(\phi) = L(\mathcal{A}_{WS1S}(\phi))$

(ii) There is an automaton $\mathcal{A}_{M2L}(\phi)$ over $\mathbb{B}^n$ such that $\mathcal{L}_{M2L}(\phi) = L(\mathcal{A}_{M2L}(\phi))$

Proof We proceed by induction over the structure of the formula ϕ. As we have seen the semantics of M2L-Str and WS1S are slightly different, also the automata $\mathcal{A}_{\mathrm{WS1S}}(\phi)$ and $\mathcal{A}_{\mathrm{M2L}}(\phi)$ are slightly different. Thus, we describe the construction of $\mathcal{A}_{\mathrm{WS1S}}(\phi)$ and to describe the construction of $\mathcal{A}_{\mathrm{M2L}}(\phi)$ we mention only the differences.

The automata for the atomic formulae $\mathsf{Succ}(X,Y)$ and $X \subseteq Y$ are depicted in Figure 2.1 and Figure 2.2 respectively. The first (respectively second) component

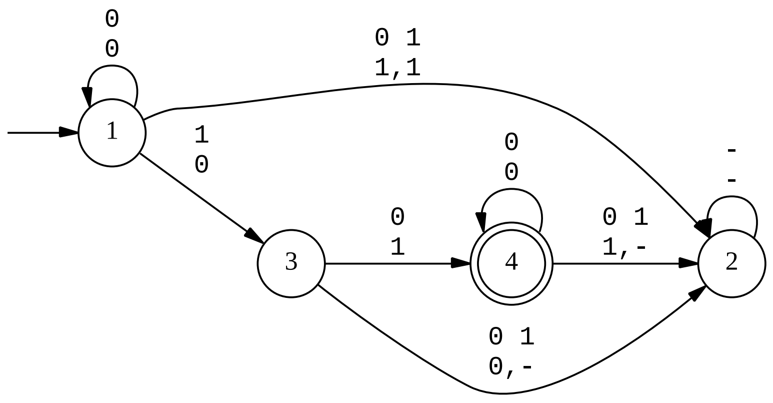


Figure 2.1: $\mathcal{A}_{\text{WS1S}}(\mathsf{Succ}(X, Y))$

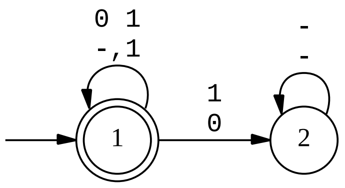


Figure 2.2: $\mathcal{A}_{\text{WS1S}}(X \subseteq Y)$

of the vectors labeling the transitions of the automata interpret the variable X (respectively Y). The symbol "-" stands for 0 and 1. For the induction step the negation $\neg$ and disjunction $\vee$ are straightforward. If $\phi = \neg\psi$, then $\mathcal{A}_{\text{WS1S}}(\phi)$ is $\overline{\mathcal{A}_{\text{WS1S}}(\psi)}$ the complement automaton of $\mathcal{A}_{\text{WS1S}}(\psi)$. If $\phi = \phi_1 \vee \phi_2$, then $\mathcal{A}_{\text{WS1S}}(\phi)$ is $\mathcal{A}_{\text{WS1S}}(\phi_1) \cup \mathcal{A}_{\text{WS1S}}(\phi_2)$ the union automaton of $\mathcal{A}_{\text{WS1S}}(\phi_1)$ and $\mathcal{A}_{\text{WS1S}}(\phi_2)$. These two cases are clear by the closure of the regular languages under complement and union. Consider now the case where $\phi = \exists X.\psi$. By induction hypothesis there is an automaton $\mathcal{A}_{\text{WS1S}}(\psi)$ over $\mathbb{B}^{n+1}$ with $L(\mathcal{A}_{\text{WS1S}}(\psi)) = L(\psi)$. The construction of $\mathcal{A}_{\text{WS1S}}(\phi)$ is done in two steps.

- First, we do a projection operation. That is, each label $(a_1, \ldots, a_{n+1}) \in \mathbb{B}^{n+1}$ occurring in a transition of $\mathcal{A}_{\text{WS1S}}(\psi)$ is replaced with $(a_2, \ldots, a_{n+1}) \in \mathbb{B}^n$. Here we suppose that the first component a_1 corresponds to variable X. We call the obtained automaton $\mathcal{A}_1$. Intuitively, $\mathcal{A}_1$ behaves similar to $\mathcal{A}_{\text{WS1S}}(\psi)$ expect that it guesses the valuation of X.

- Second, each state in $\mathcal{A}_1$ from which we can reach a final state by reading words of the form $(0, \ldots, 0)^*$ is marked as additional final state. This is necessary because the valuation of X may be longer than the valuations of the rest of the free variables. We call the obtained automaton $\mathcal{A}_2$ and we call this operation *right-quotient*. More generally, the right-quotient of a language L with a language L' is defined by

$$L/L' = \{w \mid \text{there is } u \in L' \text{ such that } wu \in L\}.$$

The automaton $\mathcal{A}_{\text{WS1S}}(\phi)$ is now the deterministic and minimal version of $\mathcal{A}_2$.

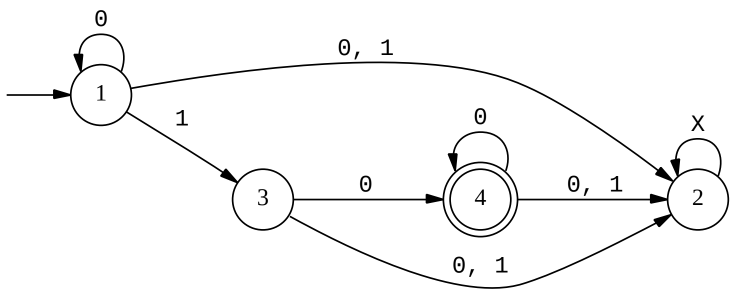
Let Π^1 be the projection function from $(\mathbb{B}^{n+1})^*$ to $(\mathbb{B}^n)^*$ defined by $\Pi^1(w) = w'$, where w' is obtained from w by replacing each vector $(a_1, \ldots, a_{n+1})$ in w by $(a_2, \ldots, a_{n+1})$. With L^1 we denote the regular language $\{(0, 0, \ldots, 0), (1, 0, \ldots, 0)\}^*$ over $\mathbb{B}^n$. We observe the following facts

$$L(\mathcal{A}_{\text{WS1S}}(\phi)) = \Pi^1(L(\mathcal{A}_{\text{WS1S}}(\psi)))/L^1 \text{ and } L(\phi) = \Pi^1(L(\psi))/L^1$$

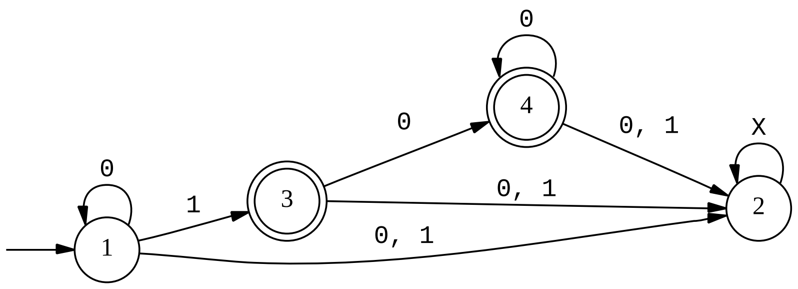
Now, by $L(\mathcal{A}_{\text{WS1S}}(\psi)) = L(\psi)$ it follows that $L(\mathcal{A}_{\text{WS1S}}(\phi)) = L(\phi)$.

For $\mathcal{A}_{\text{M2L}}(\phi)$, the second step is simply omitted. This is because in M2L-STR all valuations have the same length (the natural number k in Definition 2.4.1).

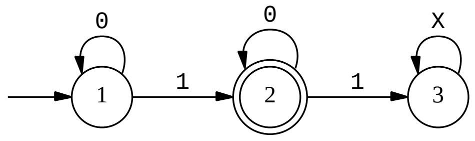
To illustrate the construction of $\mathcal{A}_{\text{WS1S}}(\phi)$ we consider the following example. Let ϕ be the formula $\exists Y. \mathsf{Succ}(X, Y)$. The automaton of the subformula $\mathsf{Succ}(X, Y)$ is displayed in Figure 2.1. The projection operation on Y applied to this automaton yields the automaton depicted in Figure 2.3(a). Note that the obtained automaton is nondeterministic. By the right-quotient operation we obtain the automaton depicted in Figure 2.3(b), in which a new final state 3 is introduced. Finally, the



(a) $\mathcal{A}_{\text{WS1S}}(\mathsf{Succ}(X,Y))$ after the projection of Y



(b) $\mathcal{A}_{\text{WS1S}}(\mathsf{Succ}(X,Y))$ after the projection and right-quotient



(c) The automaton $\mathcal{A}_{\text{WS1S}}(\exists\, Y.\, \mathsf{Succ}(X,Y))$

Figure 2.3: Several automata needed to construct $\mathcal{A}_{\text{WS1S}}(\exists\, Y.\, \mathsf{Succ}(X,Y))$

automaton of Figure 2.3(b) can be determinized and minimized to the automaton of Figure 2.3(c). In Figure 2.4, we displayed the automaton corresponding to $\exists\, Y.\, \mathsf{Succ}(X,Y)$ in M2L-STR. We can see that it is just a determinization of the automaton of Figure 2.3(a). ■

Remark 2.4.4 The right-quotient operation needed to build $\mathcal{A}_{\text{WS1S}}(\phi)$ can be car-

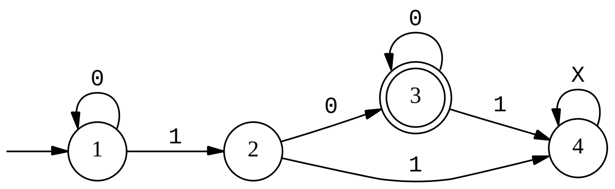


Figure 2.4: The automaton $\mathcal{A}_{\text{M2L}}(\exists Y.\, \mathsf{Succ}(X,Y))$

ried out in linear time by a breadth-first backwards search from the final states. By the above proof and from the implementation point of view, it follows that the implementation of the decision procedure for M2L-Str is simpler than the implementation of the decision procedure for WS1S.

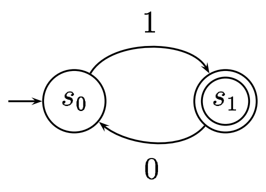
2.4.5 Connection to Regular Word Languages

As noted previously, in WS1S any word model over $\mathbb{B}^n$ can be suffixed by arbitrarily many $(0,\ldots,0) \in \mathbb{B}^n$ and the result is again a word model. Hence we explain in which sense regular languages are definable in both monadic logics, as this is not completely straightforward. Let $\Sigma = \{a_1,\ldots,a_n\}$ and let $\theta\colon \mathbb{B}^n \to \Sigma$ be the function defined by $\theta(b_1,\ldots,b_n) = a_i$, where $b_j = 1$ iff $j = i$, and let $\sim$ be the congruence relation over $(\mathbb{B}^n)^*$ defined by $u \sim v$ iff there are $i, j \in \mathbb{N}$ and $x \in (\mathbb{B}^n)^*$ with $u = x.(0,\ldots,0)^i$ and $v = x.(0,\ldots,0)^j$.We can straightforwardly extend θ to words over $(\mathbb{B}^n)^*$, sets of words, $\sim$-classes, and sets of $\sim$-classes. Now, for a regular language $L \subseteq \Sigma^*$, we can construct formulae $\phi(X_1,\ldots,X_n)$ and $\psi(X_1,\ldots,X_n)$ such that for M2L-Str we have $L = \theta(\mathcal{L}_{\text{M2L}}(\phi(\overline{X})))$ and for WS1S we have $L = \theta(\mathcal{L}_{\text{WS1S}}(\psi(\overline{X}))/\sim)$.

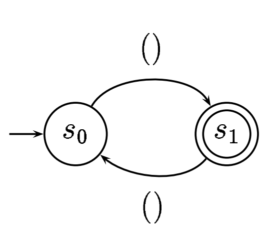
Example 2.4.1 Consider the automaton $\mathcal{A}$ depicted in Figure 2.5(a) that accepts the language $1(01)^* = \{1,\ 101,\ 10101,\ldots\}$. This language is defined by the formula

$$\begin{aligned} \mathit{alternate}_{\text{M2L}}(X) \overset{\mathit{def}}{=} \;& X(0) \wedge X(\$) \wedge & (2.2)\\ & \forall p.\; p < \$ \to (X(p+1) \leftrightarrow \neg X(p)) & (2.3) \end{aligned}$$

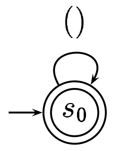
interpreted in M2L-Str. (2.2) states that the first and last positions are in X, and, by (2.3), the positions in X alternate. Observe that if we existentially quantify the variable X in $\mathit{alternate}_{\text{M2L}}(X)$, then we obtain a closed formula that is neither valid nor unsatisfiable; its corresponding automaton, given in Figure 2.5(b), is the same as $\mathcal{A}$ except its transitions are labeled with $() \in \mathbb{B}^0$.



(a) Automaton for $1(01)^*$



(b) $\exists X.\ \mathit{alternate}_{\text{M2L}}(X)$



(c) $\exists X.\ \mathit{alternate}_{\text{WS1S}}(X)$

Figure 2.5: Automata for Example

For WS1S we can define the same language by the formula

$$\begin{aligned}\mathit{alternate}_{\text{WS1S}}(X) \overset{def}{=} \exists n.\ & (\forall p.\ n < p \to \neg X(p)) \\ & X(0) \wedge X(n) \\ & \forall p.\ p < n \to (X(p+1) \leftrightarrow \neg X(p))\,.\end{aligned}$$

The only difference is that to state that n is the last position we require that X contains no positions greater than n. The language $\mathcal{L}_{\text{WS1S}}(\mathit{alternate}_{\text{WS1S}}(X))$ is $1(01)^*0^*$ and $\mathcal{L}_{\text{WS1S}}(\psi(\overline{X}))/\sim$ is $1(01)^*$. In contrast to M2L-STR, if we existentially quantify the variable X in $\mathit{alternate}_{\text{WS1S}}(X)$, then we obtain a valid formula and its automaton is depicted in Figure 2.5(c). □

2.4.6 The Logic S1S

The logic S1S is interpreted over *infinite* words. It is closely related to WS1S and differs only by allowing infinite subsets of $\mathbb{N}$ as interpretations for second-order variables. Formally, we define satisfiability in S1S, $\sigma \models_{\text{S1S}} \phi$, as follows:

Definition 2.4.5 *Satisfiability for* S1S

$$\begin{array}{lll}\sigma \models_{S1S} \mathsf{s}(t,t'), & \mathit{if} & \sigma(t') = 1 + \sigma(t) \\ \sigma \models_{S1S} X(t), & \mathit{if} & \sigma(t) \in \sigma(X) \\ \sigma \models_{S1S} \neg\phi, & \mathit{if} & \sigma \not\models_{S1S} \phi \\ \sigma \models_{S1S} \phi_1 \vee \phi_2, & \mathit{if} & \sigma \models_{S1S} \phi_1 \ \mathit{or}\ \sigma \models_{S1S} \phi_2 \\ \sigma \models_{S1S} \exists p.\ \phi, & \mathit{if} & \sigma[i/p] \models_{S1S} \phi,\ \mathit{for\ some}\ i \in \mathbb{N} \\ \sigma \models_{S1S} \exists X.\ \phi, & \mathit{if} & \sigma[M/X] \models_{S1S} \phi,\ \mathit{for\ some}\ M \in \mathsf{Pow}(\mathbb{N})\end{array}$$

A formula ϕ is *valid* in S1S (we write $\models_{\text{S1S}} \phi$) if it is satisfied by every substitution σ.

A substitution in S1S can be encoded as an infinite word and Büchi showed in [Büc62] that S1S exactly captures the ω-regular languages. In doing so, he provided an effective non-elementary transformation of S1S formulae into Büchi automata.

2.5 Monadic Second-Order Logics on Trees

Let k be a natural number. We provide here background material on WSkS. This logic is a simple generalization of WS1S from one successor to multiple (k) successors.

2.5.1 Language

The syntax of terms and formulae are given below.

$$\begin{array}{ll} t ::= \epsilon \mid p, & p \in \mathcal{V}_1 \\ \phi ::= \mathsf{s}_1(t,t) \mid \ \ldots \ \mid \mathsf{s}_k(t,t) \mid X(t) \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\, \phi \mid \exists X.\, \phi, & x, y \in \mathcal{V}_1 \text{ and } X \in \mathcal{V}_2 \end{array}$$

Terms are built from the constant ϵ and first-order variables. Formulae are built from predicates $\mathsf{s}_i(t,t')$, where $1 \leq i \leq k$ and $X(t)$ and are closed under disjunction, negation, and quantification over first-order and second-order variables. Of course, other connectives and quantifiers can be added, as is standard in classical logic.

2.5.2 Semantics

The model of WSkS has the domain $\mathcal{D} = [k]^\star$ of words (or paths) w over $[k]$, where k is the number of successors. Substitutions σ assign words $w \in \mathcal{D}$ to first-order variables p and finite sets $P \subseteq \mathcal{D}$ of words to second-order variables. The constant ϵ denotes the empty word. The denotation of s_i (called the i-th successor relation and written S_i) is concatenation with the letter i; *i.e.* for $w \in \mathcal{D}$, $(w, w.i) \in S_i$, for $i \leq k$. The formula $X(t)$ is true when the position denoted by t is in the set denoted by X. We write $\mathcal{D} \models_{\text{WSkS}} \phi$ to say that the WSkS formula ϕ is valid: $\mathcal{D}, \sigma \models_{\text{WSkS}} \phi$ holds for all substitutions σ under the given interpretation.

Definition 2.5.1 *Satisfiability for* WSkS

$$
\begin{array}{lll}
\mathcal{D},\sigma \models_{\mathit{WSkS}} \mathsf{s}_i(t,t'), & \mathit{if} & \sigma(t') = \sigma(t).i,\ \mathit{for}\ i \in [k] \\
\mathcal{D},\sigma \models_{\mathit{WSkS}} X(t), & \mathit{if} & \sigma(t) \in \sigma(X) \\
\mathcal{D},\sigma \models_{\mathit{WSkS}} \neg\phi, & \mathit{if} & \mathcal{D},\sigma \not\models_{\mathit{WSkS}} \phi \\
\mathcal{D},\sigma \models_{\mathit{WSkS}} \phi_1 \vee \phi_2, & \mathit{if} & \mathcal{D},\sigma \models_{\mathit{WSkS}} \phi_1\ \mathit{or}\ \mathcal{D},\sigma \models_{\mathit{WSkS}} \phi_2 \\
\mathcal{D},\sigma \models_{\mathit{WSkS}} \exists p.\ \phi, & \mathit{if} & \mathcal{D},\sigma[i/p] \models_{\mathit{WSkS}} \phi,\ \mathit{for\ some}\ i \in \mathcal{D} \\
\mathcal{D},\sigma \models_{\mathit{WSkS}} \exists X.\ \phi, & \mathit{if} & \mathcal{D},\sigma[M/X] \models_{\mathit{WSkS}} \phi,\ \mathit{for\ some\ finite}\ M \in \mathsf{Pow}(\mathcal{D})
\end{array}
$$

In order to clarify the terminology of "logic over trees" for WSkS, we mention that the pair $\langle \mathcal{D}, \sigma \rangle$ can be encoded by a k-ary tree t. Let ϕ be a formula with the free second-order variables $X_1, \ldots, X_n$. If $\sigma(X_i)$ is empty for all $i \leq n$, then t is the empty tree. Otherwise, let m be the maximal length in σ, *i.e.* $\mathsf{max}\{|w| \mid w \in \sigma(X_i) \text{ for } i \leq n\} \leq m$. The set of nodes of t is $\{w \in \mathcal{D} \mid |w| \leq m|\}$. The nodes are labeled by elements of $\mathbb{B}^n$. The i-th component of the label of the node w is 1 iff w is equal to or contained in the value of the i-th variable, *i.e.* $w = \sigma(p)$ or $w \in \sigma(X)$, respectively. One sometimes then writes $\underline{t} \models \phi$ instead of $\mathcal{D}, \sigma \models_{\mathrm{WSkS}} \phi$. We define $\mathcal{L}_{\mathrm{WSkS}}(\phi)$ as the set $\{t \mid \underline{t} \models_{\mathrm{WSkS}} \phi\} \subseteq T_{\mathbb{B}^n}$. If $L = \mathcal{L}_{\mathrm{WSkS}}(\phi)$ for some ϕ, then L is called *definable* in WSkS.

Example 2.5.1 The following is a simple example of a specification in WS2S.

$$X(\epsilon)\ \wedge\ \forall p.\, X(p.0) \Leftrightarrow X(p.1)\ \wedge\ \forall p.\, \neg Y(p.0) \vee \neg Y(p.1)\,.$$

This states that X contains the root position ϵ and that a position p is in X iff its brother is also in X. Moreover, for any p, Y contains at most one of p's successors. The interpretation $\{\epsilon, 0, 1, 00, 01\}$ for X and $\{0, 01, 11\}$ for Y satisfies this formula and is encoded by the following tree where the upper component of each pair encodes the interpretation of X and the lower encodes Y.

$$\begin{tikzcd}
 & & & \begin{pmatrix}1\\0\end{pmatrix} \arrow[dll] \arrow[drr] & & & \\
 & \begin{pmatrix}1\\1\end{pmatrix} \arrow[dl] \arrow[dr] & & & & \begin{pmatrix}1\\0\end{pmatrix} \arrow[dl] \arrow[dr] & \\
\begin{pmatrix}1\\0\end{pmatrix} & & \begin{pmatrix}1\\1\end{pmatrix} & & \begin{pmatrix}0\\0\end{pmatrix} & & \begin{pmatrix}0\\1\end{pmatrix}
\end{tikzcd}$$

□

Decidability

Doner [Don65] and Thatcher and Wright [TW68] proved that the set of definable languages in WSkS coincides with the set of regular k-ary tree languages. Büchi [Büc60] and Doner [Don65] have applied concepts of generalized finite tree automata to decide WSkS. These used concepts parallel to those developed by Büchi and Elgot for the one-successor case.

2.5.3 Connection to Regular Tree languages

Similarly to what we have already noticed for WS1S in Section 2.4.5, if a tree t is a model of a formula ϕ in WSkS then any tree t' obtained from t by adding new nodes that are labeled with $(0,\ldots,0) \in \mathbb{B}^n$ is also a model of ϕ. If we adapt the congruence relation $\sim$ to k-ary trees, then we can also prove that a regular tree language L over Σ is definable in WSkS iff there is formula ψ such that $L = \theta(\mathcal{L}_{\mathrm{WSkS}}(\psi)/\sim)$, where θ is the substitution defined in Section 2.4.5. We can also generalize M2L-Str to M2L-k-Tree and analogously obtain the result that a regular tree language L over Σ is definable in M2L-k-Tree iff there is formula ψ such that $L = \theta(\mathcal{L}_{\mathrm{M2L\text{-}k\text{-}tree}}(\psi))$.

2.6 Proof Tools for some Monadic Logics

Despite the high complexity of WS1S, several research groups have implemented proof tools for this logic [KM01, MBBC95, MC97] that work surprisingly well on many non-trivial problems. The Mona system is one of the most well known and most widely used tool. Besides the decision procedure of WS1S, it also implements the decision procedures of M2L-Str, WS2S, and M2L-Tree[3]. Mona follows the automata construction described in the proof of Theorem 2.4.3: it translates formulae of monadic second-order logic to deterministic minimal finite automata. A valid formula is particularly simple to recognize: its corresponding automaton is the so called trivial automaton which consists only of one state that is both the initial state and the single final state and it has a self loop as transition for every alphabet symbol. Invalid formulae have non-trivial automata. Mona is able to extract from a non-trivial automaton of a formula ϕ a minimal length word that defines a counter-example for ϕ.

Mona uses BDDs [Bry92, Bry86] in order to compress the representation of the transition function of automata. This representation of automata is minimal in two ways: BDDs are reduced to their canonical form and the transition function and the state space represented are those of the minimal automaton. Minimality of BDDs

[3]M2L-Tree stands for M2L-2-Tree (see Section 2.5.3)

is preserved automatically by the algorithms that calculate them. Minimization of the automaton is enforced using an algorithm that is quadratic in the size of the automaton. For a more detailed treatment of this issue see [HJJ+96].

MONA syntax is essentially that of Section 2.4.1 in the word case and of Section 2.5.1 in the tree case, augmented with syntactic sugar. A MONA specification consists of a mode declaration, that is the specification of the logic to be used, followed by a sequence of predicates. At the end of the program a formula which has to be decided is specified.

To illustrate how MONA works, we consider the following simple example. We define predicates Odd and Even as follows in order to prove that every set can be partitioned in a set containing only even numbers and a set containing only odd numbers.

ws1s;

pred $\mathrm{Odd}(\textsf{var2}\ S) = \exists O.\, S\ \textsf{Sub}\ O \wedge \neg O(0) \wedge O(1) \wedge \forall p.\, O(p+2) \Rightarrow O(p)$;

pred $\mathrm{Even}(\textsf{var2}\ S) = \exists E.\, S\ \textsf{Sub}\ E \wedge E(0) \wedge \neg E(1) \wedge \forall p.\, E(p+2) \Rightarrow E(p)$;

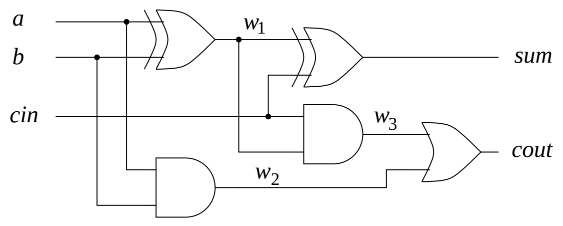
$\forall S.\, \exists U, V.\, \mathrm{Odd}(U) \wedge \mathrm{Even}(V) \wedge \forall p.\, S(p) \Leftrightarrow (U(p) \Leftrightarrow \neg V(p))$;

The keyword ws1s indicates that we use the logic WS1S. The predicate Sub stands for the subset relation. The keyword pred starts a predicate definition. In our example two new predicates Odd and Even are defined. They both take a second-order term S as an argument, which is indicated by the parameter declaration var2 S. When invoking MONA with the above declarations, we obtain the following answer which states the validity of the submitted formula.

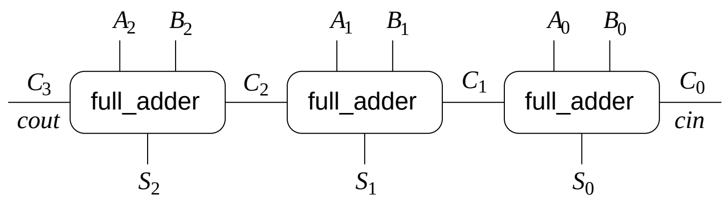
```
ANALYSIS: Formula is valid
```

2.7 Applications

So far we have introduced regular languages, finite automata, and monadic logics and showed their relationship to each other. In this section we want to address the practical use of these mathematical concepts. The monadic logics offer the possibility of modeling and reasoning about hardware and software systems. The use of monadic logics on words (M2L-STR and WS1S) in modeling combinational and sequential systems as well as in modeling protocols has already been demonstrated (see [BF98, BK98]). In our exposition here, we use WS1S and WS2S to reason about linear-structured and tree-structured systems.



(a) 1-bit full adder



(b) An n-bit adder instance, for $n = 3$

Figure 2.6: Ripple-Carry Adder Circuit

2.7.1 Example: Parameterized Ripple-Carry Adder

In this section, we show how WS1S can be used to specify and verify a family of ripple-carry adders. Let us start with modeling the adder circuit and show first how to model a 1-bit full adder. In WS1S, we can formalize circuits as relations over their external ports. Circuits are built from relations representing primitives, such as transistors or gates, and are combined with conjunction and "wired together" using existential quantification. This style of representation is standard in the theorem proving community and scales well to complex systems [CGM86].

For example, to model the 1-bit full adder given in Figure 2.6(a) we begin by defining the following gates.

$$\begin{aligned} \mathsf{and}(a, b, o) &\equiv o \leftrightarrow (a \wedge b) \\ \mathsf{or}(a, b, o) &\equiv o \leftrightarrow (a \vee b) \\ \mathsf{xor}(a, b, o) &\equiv o \leftrightarrow ((a \wedge \neg b) \vee (\neg a \wedge b)) \end{aligned}$$

We then compose these to model the adder circuit. The top half of the adder consists of two *xor*-gates, connected by an internal wire w_1, which compute the sum bit *out*. The bottom half uses the internal wire w_1, as well as the two inputs a and b, to compute the carry-out bit *cout*. Our definition conjoins the gate descriptions and projects away the internal wires:

$$\begin{aligned}\mathsf{full\_adder}(a, b, out, cin, cout) \equiv \exists w_1, w_2, w_3.&\\ &\mathsf{xor}(a, b, w_1) \wedge \mathsf{xor}(w_1, \mathrm{cin}, \mathrm{out}) \wedge\\ &\mathsf{and}(a, b, w_2) \wedge \mathsf{and}(\mathrm{cin}, w_1, w_3) \wedge \mathsf{or}(w_3, w_2, \mathrm{cout})\end{aligned}$$

Figure 2.6(b) suggests how an adder can be iterated, in a "ripple-carry" way, to construct an n-bit adder, for $n = 3$. In general, we can model an n-bit adder by wiring together n 1-bit full adders where the carry-out of the ith adder is the carry-in of the i+1st. The first carry has the value of the carry-in and the last has the value of the carry-out.

It is easy to formalize a ripple-carry architecture by a WS1S formula. If we use C to represent the sequence of carries, we can formalize the general case by the following formula, which relates three strings (the inputs A and B and the output S) and two Booleans (the carry-in cin and carry-out cout); the number of bits added is given by the parameter n.

$$\begin{aligned}&\mathsf{adder}(n, A, B, \mathrm{S}, \mathrm{cin}, \mathrm{cout}) \equiv\\ &\quad \exists C.\, (C(\mathsf{0}) \leftrightarrow cin) \wedge (C(n) \leftrightarrow \mathrm{cout}) \wedge\\ &\qquad \forall p.\, p < n \rightarrow\\ &\qquad\quad \mathsf{full\_adder}(A(p), B(p), S(p), C(p), C(p+1))\end{aligned}$$

This is a direct formalization of our natural language description where quantification over positions p formalizes iteration (generalized conjunction) over the n 1-bit full adders.

Let us now turn our attention to the specification of the adder in which we state how strings, representing n-bit binary numbers, are added. We model addition by formalizing the standard algorithm for adding base-two numbers: The ith output bit is set if the sum of the ith inputs and carry-in is 1 mod 2, and the ith carry bit is set if at least two of the previous inputs and carry-in are set. Since we specify the behavior of an n-bit adder, we must restrict the addition modulo 2^n and compute the carry values as special cases.

$$\begin{aligned}\mathsf{at\_least\_two}(a, b, c, d) &\equiv d \leftrightarrow (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)\\ \mathsf{mod\_two}(a, b, c, d) &\equiv a \leftrightarrow b \leftrightarrow c \leftrightarrow d\end{aligned}$$

$$\begin{aligned}&\mathsf{spec}(n, A, B, \mathrm{S}, \mathrm{cin}, \mathrm{cout}) \equiv\\ &\quad \exists C.\, (C(\mathsf{0}) \leftrightarrow cin) \wedge (C(n) \leftrightarrow \mathrm{cout})\\ &\qquad \forall p.\, p < n \rightarrow\\ &\qquad\quad \mathsf{at\_least\_two}(A(p), B(p), C(p), C(p+1))\\ &\qquad\quad \mathsf{mod\_two}(A(p), B(p), S(p), C(p))\end{aligned}$$

This specifies a language over $\mathbb{B}^6$, which encodes interpretations for n, A, B, S, *cin* and *cout*. For example, one string in this language is:

	0	1	2	3	4
n	0	0	0	0	1
A	1	1	0	0	0
B	1	0	0	1	0
S	1	0	1	1	0
cin	1	0	0	0	0
cout	0	0	0	0	0

Recall that in the interpreting string for first-order variables like n, exactly the nth bit is 1. Moreover, *cin* and *cout* are true iff the 0th bit of their interpreting string is 1. Hence the above encodes that the bit-width $n = 4$. Reading Boolean strings as binary numbers from left to right (up to the nth position) the next three lines encode $A = 3$, $B = 9$, and $S = 13$. Since the carry-in is set and the carry-out is not, this string is indeed in the addition relation modeled.

Verification

Now having both the circuit and the specification of the ripple-carry adder, we can then formalize its correctness as follows.

$$\begin{aligned}&\forall n.\, \forall A, B, S.\, \forall cin, cout.\\ &\quad \mathsf{adder}(n, A, B, S, cin, cout) \leftrightarrow\\ &\quad \mathsf{spec}(n, A, B, S, cin, cout)\end{aligned}$$

This formula is proven valid by Mona in under a second.

2.7.2 Example: Parameterized Carry-Lookahead Adder

In this section, we consider a family of carry-lookahead adders, which are tree-like structured circuits. We begin by describing a family of n-bit *carry-lookahead adder*s (or Clas) whose overall structure is given (for $n = 4$) in Figure 2.7. We do not describe the adder in detail (see [CLR92]) and restrict ourselves to a few comments. The adder operates in two phases: an *upward* phase and *downward* phase. In the upward phase, a carry status is computed for each internal node. This indicates whether an incoming carry is *killed*, *propagated*, or a new carry is *generated*. The carry status is given by the wires E_1 and E_2. At the leafs of the tree, the carry status for each digit is computed using the circuit depicted in Figure 2.8(a). At inner nodes, an operator $\otimes$ is used to combine the carry statuses of the successor nodes (see Figures 2.8(b) and 2.8(d)).

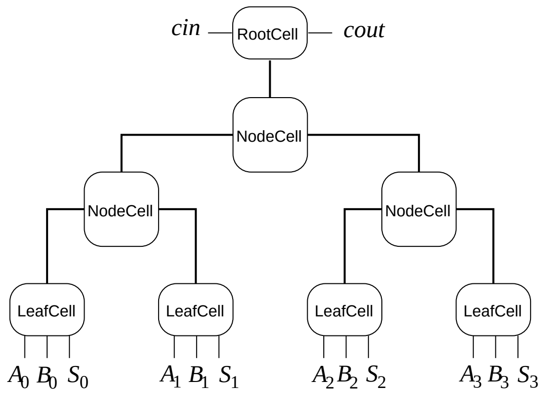


Figure 2.7: An n-bit Cla instance, for $n = 4$

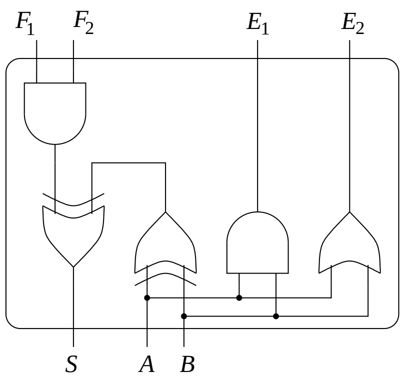
In the downward phase, an inner node passes the incoming carry bits F_1 and F_2 unchanged as signals F_1^0 and F_2^0 to its left successor; the carry bits F_1^1, F_2^1 for the right successor are computed using the operator $\otimes$ from the carry statuses and the incoming carry bits (see Figure 2.8(d)). As indicated in Figure 2.8(c), the incoming carry at the root is given by *cin* and the outgoing carry by *cout*. At the leaf nodes, the incoming carry bits F_1 and F_2 are used to compute the sum bit as depicted in Figure 2.8(a).

Modeling the Circuits of the CLAs

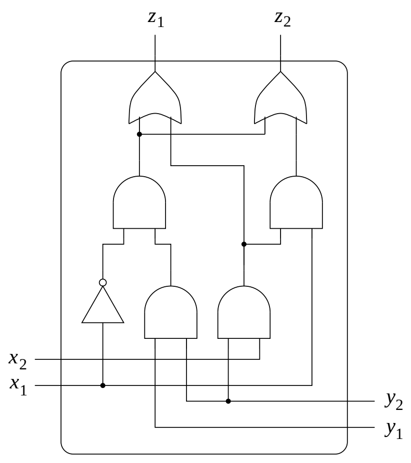
We translate the circuits in Figures 2.8(a)–2.8(d) directly into the following WS2S formulae. Note that LeafCell and NodeCell take an extra argument p that indicates which leaf or node values are used.

$$\begin{aligned}
&\mathsf{pred\ LeafCell}(\mathsf{var2}\ A, B, S, F_1, F_2, E_1, E_2, \mathsf{var1}\ p) = \\
&\quad \mathsf{and}(A(p), B(p), E_1(p)) \ \wedge\ \mathsf{or}(A(p), B(p), E_2(p)) \wedge \\
&\quad \exists w_1, w_2.\ \mathsf{and}(F_1(p), F_2(p), w_1)) \wedge \\
&\qquad\qquad \mathsf{xor}(A(p), B(p), w_2) \ \wedge\ \mathsf{xor}(w_1, w_2, S(p))
\end{aligned}$$

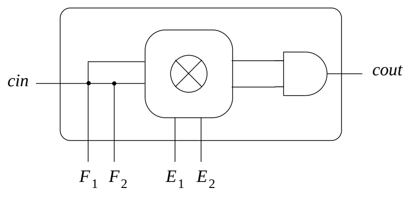
$$\begin{aligned}
&\mathsf{pred\ NodeCell}(\mathsf{var2}\ F_1, F_2, E_1, E_2, \mathsf{var1}\ p) = \\
&\quad (F_1(p.0) \Leftrightarrow F_1(p)) \ \wedge\ (F_2(p.0) \Leftrightarrow F_2(p)) \wedge \\
&\quad \mathsf{op}(F_1(p), F_2(p), E_1(p.0), E_2(p.0), F_1(p.1), F_2(p.0)) \wedge \\
&\quad \mathsf{op}(E_1(p.0), E_2(p.0), E_1(p.1), E_2(p.1), E_1(p), E_2(p))
\end{aligned}$$



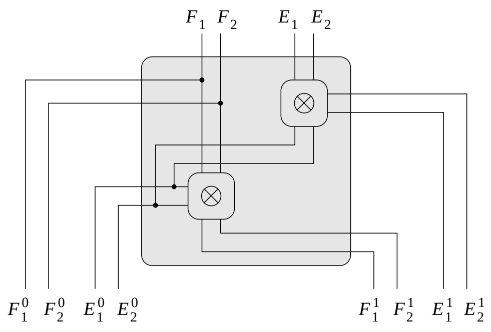
(a) LeafCell Circuit



(b) $\otimes$ Circuit



(c) RootCell Circuit



(d) NodeCell Circuit (op)

Figure 2.8: Components of the Cla Circuit

$$\begin{aligned}
&\mathsf{pred\ RootCell}(\mathsf{var2}\ F_1, F_2, E_1, E_2, \mathsf{var0}\ \mathit{cin}, \mathit{cout}) \equiv \\
&\quad (\mathit{cin} \Leftrightarrow F_1(\epsilon)) \ \wedge\ (\mathit{cin} \Leftrightarrow F_2(\epsilon)) \wedge \\
&\quad (\exists a, b.\, \mathsf{op}(F_1(\epsilon), F_2(\epsilon), E_1(\epsilon), E_2(\epsilon), a, b) \wedge \\
&\qquad\qquad \mathsf{and}(a, b, \mathit{cout}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{pred\ op}(\mathsf{var0}\ x_1, x_2, y_1, y_2, z_1, z_2) = \exists w_1, w_2, w_3, w_4, w_5. \\
&\ \ \mathsf{not}(x_1, w_1) \ \wedge\ \mathsf{and}(y_1, y_2, w_2) \ \wedge\ \mathsf{and}(x_2, y_2, w_3) \wedge \\
&\ \ \mathsf{and}(w_1, w_2, w_4) \ \wedge\ \mathsf{and}(x_1, w_3, w_5) \wedge \\
&\ \ \mathsf{or}(w_4, w_5, z_1) \ \wedge\ \mathsf{or}(w_4, w_3, z_2)
\end{aligned}$$

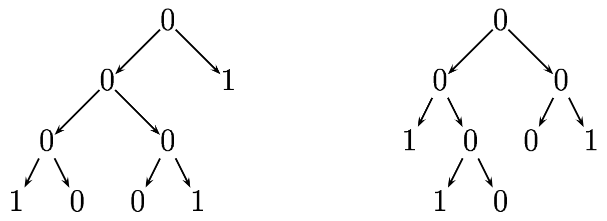
The overall circuit, given in Figure 2.7, is modeled by the predicate

$$\begin{aligned}
&\mathsf{pred\ cla}(\mathsf{var2}\ A, B, S, \mathsf{var0}\ \mathit{cin}, \mathit{cout}) = \exists T, E_1, E_2, F_1, F_2. \\
&\ \mathsf{RootCell}(F_1, F_2, E_1, E_2, \mathit{cin}, \mathit{cout}) \wedge \\
&\ (\forall p.\, (\mathsf{leaf}(p, T) \Rightarrow \mathsf{LeafCell}(A, B, S, F_1, F_2, E_1, E_2, p)) \wedge \\
&\qquad (\mathsf{node}(p, T) \Rightarrow \mathsf{NodeCell}(F_1, F_2, E_1, E_2, p))) \wedge \\
&\ \mathsf{shape\_cond}(A, B, S, T)\,.
\end{aligned}$$

RootCell initializes the carry-in bit and computes the carry-out bit. The next two lines correspond to a for-loop with discrimination (pattern matching) for each position p. The first implication gives the base case: each leaf of the circuit is LeafCell. The second gives the step case: each inner node of the circuit is NodeCell. The predicate shape_cond fixes the shape of the inputs; we explain this in the following section.

Specification

In WS1S we encoded numbers as bit-words. In WS2S we have binary trees and encode numbers using the labels on a tree's frontier. For example, the following trees represent the bit-words 10011 and 11001.



An important requirement in specifying tree structured adders, is that, once we fix the format of the adder to be a particular shape, both inputs and the output must also be of that shape; that is, numbers must be encoded at leaf nodes in the same positions in these three trees. This kind of requirement, which corresponds roughly to a kind of "type-correctness" for the inputs, is easy to specify in a high-level programming or specification language where recursive data-types can formalize this type (or "shape") constraint. In our setting, we define a predicate (shape_cond, which we used in the specification of cla above) that enforces this requirement.

We proceed in several steps. First, we characterize those trees with a particular shape as those T where:

(i) T is not empty.

(ii) T is closed under the parent relation: if a position p is in T then its parent, denoted by $p\uparrow$, is also in T.

(iii) If an inner node p is in T then both its successors $p.0$ and $p.1$ are in T too.

This "type" is formalized by

$$\begin{aligned}\mathsf{pred\ shape}(\mathsf{var2}\ T) = \ & (\exists p.\, T(p)) \wedge \\ & \forall p.\, (T(p) \Rightarrow T(p\uparrow)) \ \wedge (T(p) \ \wedge \ T(p.0) \Rightarrow T(p.1)) \wedge \\ & (T(p) \ \wedge \ T(p.1) \Rightarrow T(p.0))\,.\end{aligned}$$

Second, we say that a tree X has shape T (X is of type T) if all its positions constitute a subset of the leafs of T.

$$\mathsf{pred\ is\_of\_shape}(\mathsf{var2}\ X, T) = \forall p.\, X(p) \Rightarrow \mathsf{leaf}(p, T)\,.$$

Finally, we combine these to formalize shape_cond that holds when the trees A, B, and S, which represent base-two numbers, have the same shape T.

$$\begin{aligned}\mathsf{pred\ shape\_cond}(\mathsf{var2}\ A, B, S, T) = \ & \mathsf{shape}(T) \ \wedge \ \mathsf{is\_of\_shape}(A, T) \wedge \\ & \mathsf{is\_of\_shape}(B, T) \ \wedge \ \mathsf{is\_of\_shape}(S, T)\,.\end{aligned}$$

To complete our specification, we define several auxiliary predicates in Figure 2.9 that allow us to traverse the leaves of a valid input tree from left to right. For a tree T satisfying the shape predicate and a position p, $\mathsf{first}(p, T)$ checks if p is the left-most leaf in T and $\mathsf{last}(p, T)$ checks if p is the right-most leaf in T. The predicate $\mathsf{next}(p, q, T)$ checks if p and q are leaves in T and q is the next leaf to the right of q. Using these, our specification can be defined as follows.

$$
\begin{aligned}
&\mathsf{pred\ leaf}(\mathsf{var1}\ p, \mathsf{var2}\ T) = T(p)\ \wedge\ \neg T(p.0)\ \wedge\ \neg T(p.1)\\
&\mathsf{pred\ node}(\mathsf{var1}\ p, \mathsf{var2}\ T) = T(p)\ \wedge\ T(p.0)\ \wedge\ T(p.1)\\
&\mathsf{pred\ path}(\mathsf{var1}\ p, \mathsf{var2}\ X) = \mathsf{leaf}(p, X) \wedge\\
&\quad \forall x.\, (X(x) \Rightarrow X(x\!\uparrow))\ \wedge\ (X(x.0) \Rightarrow \neg X(x.1))\\
&\mathsf{pred\ next}(\mathsf{var1}\ p, q, \mathsf{var2}\ T) = p \neq q\ \wedge\ \mathsf{leaf}(p, T)\ \wedge\ \mathsf{leaf}(q, T) \wedge\\
&\quad \exists P, Q.\, \mathsf{path}(p, P)\ \wedge\ \mathsf{path}(q, Q) \wedge\\
&\quad \exists s.\, \exists S.\, P(s)\ \wedge\ Q(s)\ \wedge\ P(s.0) \wedge\\
&\qquad\qquad Q(s.1)\ \wedge\ \mathsf{path}(s, S) \wedge\\
&\quad \forall u.\, (P(u.0)\ \wedge\ u \neq s \Rightarrow S(u.0)) \wedge\\
&\quad\ (Q(u.1)\ \wedge\ u \neq s \Rightarrow S(u.1))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{pred\ first}(\mathsf{var1}\ p, \mathsf{var2}\ T) = \mathsf{leaf}(p, T)\ \wedge\ \exists X.\, \mathsf{path}(p, X) \wedge\\
&\forall u.\, X(u.1) \Rightarrow (\forall v.\, u.0 \leq v \Rightarrow \neg T(v))\\
&\mathsf{pred\ last}(\mathsf{var1}\ p, \mathsf{var2}\ T) = \mathsf{leaf}(p, T)\ \wedge\ \exists X.\, \mathsf{path}(p, X) \wedge\\
&\quad \forall u.\, X(u.0) \Rightarrow (\forall v.\, u.1 \leq v \Rightarrow \neg T(v))
\end{aligned}
$$

Figure 2.9: Auxiliary predicates for the Cla

$$
\begin{aligned}
&\mathsf{pred\ spec}(\mathsf{var2}\ A, B, S, \mathsf{var0}\ \mathit{cin}, \mathit{cout}) = \exists T, C.\\
&\quad \mathsf{shape\_cond}(A, B, S, T) \wedge\\
&\quad \forall p.\, \mathsf{leaf}(p, T) \Rightarrow \mathsf{mod\_two}(A(p), B(p), C(p), S(p)) \wedge\\
&\quad \exists p.\, \mathsf{first}(p, T)\ \wedge\ (\mathit{cin} \Leftrightarrow C(p)) \wedge\\
&\quad \exists p.\, \mathsf{last}(p, T)\ \wedge\ \mathsf{at\_least\_two}(A(p), B(p), C(p), \mathit{cout}) \wedge\\
&\quad \forall p, q.\, \mathsf{leaf}(p, T)\ \wedge\ \mathsf{next}(p, q, T) \Rightarrow\\
&\qquad\qquad \mathsf{at\_least\_two}(A(p), B(p), C(p), C(q))\ .
\end{aligned}
$$

Verification

We can now verify that the circuits of the Clas satisfy the specification, *i.e.* that a Cla of any size actually adds its inputs. We formalize this as

$$
\begin{aligned}
&\forall A, B, S.\, \forall \mathit{cin}, \mathit{cout}.\\
&\mathsf{cla}(A, B, S, \mathit{cin}, \mathit{cout}) \Leftrightarrow \mathsf{spec}(A, B, S, \mathit{cin}, \mathit{cout})\,.
\end{aligned}
$$

This formula is proved by MONA in one second.

2.8 Chapter Summary

More than forty years ago the monadic logics were introduced. They are based on finite automata theory and are proved to be decidable. Because of their high complexity, they appeared for long time to be impracticable. Lastly, through the implementation of their decision procedures in the proof tools MONA, STEP, and Mosel, which have been successfully applied to several problems, the monadic logics have attracted a considerable importance in practice.

As illustrated through our carry-lookahead adder example, problems must be modeled at too low a level of abstraction: in WS2S everything must be mapped onto binary branching Boolean trees. This can make models difficult to design and understand (a witness is the concern with "shape constraints" in Section 2.7.2) and lead to modeling errors. We work on building a high-level specification language that eliminates this drawback. We develop a language that allows users to specify tree languages in WS2S using data-types like those found in modern programming languages. Our new specification language is the subject of the next chapter.

Chapter 3

Lisa: A Specification Language Based on WS2S

*In the previous chapter we have demonstrated how monadic logics (*M2L-Str*,* WS1S *and* WS2S*) can be used for system verification. We have argued that the languages provided by these logics can not be easily used as specification languages as they lack of high-level programming concepts like types. The objective of this chapter is to introduce a new specification language without this deficiency.*

We now integrate two concepts from programming languages into a specification language based on WS2S*, namely high-level data structures such as records and recursively defined datatypes. Our integration is based on a new logic whose variables range over record-like trees and an algorithm for translating datatypes into tree automata. We have implemented* Lisa*, a prototype system based on these ideas, which, when coupled with a decision procedure for* WS2S *like the* Mona *system, results in a verification tool that supports both high-level specifications and complexity estimations for the running time of the decision procedure.*

3.1 Introduction

There is a large number of research groups [HJJK95, KMMG97, KMMP97, MBBC95, MC97] that have implemented verification tools based on a decision procedure for WS1S and WS2S successors. Experience, *cf.* [BK95], indicates that although such tools are powerful aids to verification, their usefulness is limited by two major problems. First, the specification language is low-level; writing specifications in WS2S is an experience akin to programming in assembly language. Second, the complexity of verification is very high; WS2S and related monadic logics are amongst the most expressive decidable logics known, but one pays the price that the decision problem requires non-elementary time, which is a strong practical limitation.

In this chapter, we propose an approach that addresses both problems. Our contributions are both theoretical and conceptual. Our theoretical contributions are

(1) to define a logic whose formulae define relations between record-like trees ("feature trees"). These relations are encoded by WS2S formulae and, thus, recognized by tree automata. This logic forms the kernel of a specification language whose decision procedure is based on that of WS2S. Our logic comes with its own interpretation domain (*i.e.* the trees) and interpretation function. This distinguishes it from notation (or macros) whose semantics is defined by syntactic translation (or unfolding).

(2) We describe explicitly the direct translation of the part of the logic in which one defines datatypes to deterministic tree automata via alternating tree automata. We show that in many practical cases this is polynomial time computable, and exponential in the worst case.

Our conceptual contribution is to propose an approach that simultaneously addresses the two main limitations of WS2S. The base logic of feature trees, combined with recursive types, provides a formalism for high-level abstract specification. In particular, there is direct support for formalizing record-like data-structures, *e.g.* accessing subtrees via symbolic keywords, which are supported in most modern programming languages. Moreover, types provide a handle on the complexity of the decision procedure. Types are directly translated to tree automata (as opposed to indirectly via an initial translation to WS2S formulae) and, as noted above, we bound the complexity of this process.

We have motivated our combination by arguing that it alleviates many of the problems of specification and verification with WS2S. An alternative way to approach and understand our proposal is by comparison with standard programming languages. Early programming languages, like assembly languages, Lisp, and Fortran, provided little or no support for datatypes. The user encoded data explicitly in memory. This is analogous to WS2S where the only primitive "type" supported is sets of positions in the binary tree. Hence, the user must laboriously encode other kinds of data, say k-ary trees whose nodes are labeled from some finite set, in terms of unlabeled binary trees. As with programming in assembler, this is possible, but not recommended, and the result falls far short of constituting a comprehensible specification. More advanced programming languages, like ML, provide means of abstractly formalizing data using type declarations. This is important also for structuring the program: these declarations are part of the program and integrate a specification language into a programming language (which is also a specification language) in a controlled and natural way. This is analogous to types in our proposal; types structure the specification and interact with defined predicates by restricting the scope of quantification to elements of the defined types (relativized quantification).

One important way in which the programming analogy breaks down is that datatypes in our specification language (compared to datatypes in, say, ML) have the same expressiveness as the full language. Both define tree automata and hence both are "WS2S complete" in the sense that they can define any WS2S definable relation. However, they do differ from formulae in the full logic in succinctness by a non-elementary factor. Said the other way round, a specification using types may trade verbosity for a gain in efficiency. Since we have found that one uses types often in a specification, it is important to give the user a means to control the cost of the usage of types, at least to some degree. Therefore we give the translation procedure of types explicitly. Translation procedures have been proposed for various kinds of regular systems of equations over words and trees [Ard61, BL80, GS84], but none of these is applicable to type systems as rich as ours. The principal distinction is that our type definitions support conjunctions of types, which is natural in our logic where subtree positions (record-fields) are accessed by atomic formulae. We establish a relationship between such type definitions and alternating top-down tree automata.

Although we see the contributions in this chapter as theoretical and conceptual, their ultimate validation must be empirical. We have implemented and tested our ideas. The base logic and type system are implemented in a prototype system called LISA, which is coupled with the MONA system. We have used LISA to carry out several case studies, one of which we report on here.

The chapter is organized as follows: in Section 3.1.1 we motivate the LISA approach and in Section 3.1.2 we introduce LISA informally. Section 3.2 introduces the kernel language of LISA and Section 3.4 gives its compilation into WS2S. In Section 3.4 we describe the LISA type system: the syntax and semantics, the relationship to systems of language equations, and the compilation into tree automata. In Section 3.5 we show an extension of the LISA type system that has only a linear blow-up in the size of the resulting tree automata. In Section 3.6 we study the case where the LISA feature trees can be encoded as words. We will call this fragment LLISA and report on an example done in LLISA. We report on related work in Section 3.7 and, finally, we draw conclusion in Section 3.8.

3.1.1 Starting Point

Our work is directly inspired by the work of Klarlund and Schwartzbach on a system called FIDO [KNS96]; FIDO is based on the idea that one can encode the values of any fixed finite set and write finite-domain constraints in WS2S. FIDO deserves the credit of being the first approach to integrating programming language concepts with

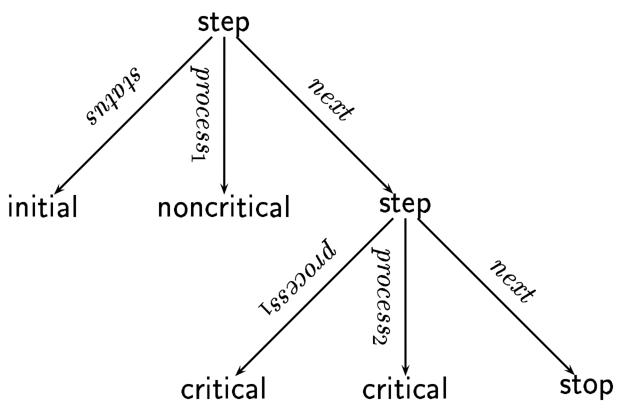


Figure 3.1: A feature tree

MONA. Our work started with the study of FIDO; we observed that the only data of interest are record-modeling trees and that, under this view, the expressiveness of FIDO's datatypes is the full expressiveness of FIDO. Moreover, FIDO was conceived and explicitly described as a "programming notation"; its semantics was defined by compilation into MONA, the assembly language. We felt that this did not take the programming-language point of view all the way. There, one abstracts away from the underlying machine model (be it jumps or sets of positions) and defines a new calculus/logic with its own semantics; then one can prove the compilation correct. The new logic should be small and simple; it forms the kernel of the language, which itself may be rich in notation. Regarding trees and datatypes in FIDO: These were used mainly to define "domains" for position variables. The type declarations for finite-domain values in FIDO are expressed in LISA by non-recursive datatype definitions (denoting finite sets of trees); this is yet another example indicating the advantage (in conceptual simplicity) of having trees as the interpretation domain.

The idea of using feature trees to model records stems from [AKPS94]. The first-order logic over feature trees is decidable in non-elementary time [BS93, Vor96] and to our knowledge, no decision procedure has been implemented yet. The basic relation in that logic, besides the unary label relation that corresponds to $l(t, \epsilon)$, is the direct-subtree relation $f(t, t')$. The addition of that relation to our logic would make the validity test undecidable. It is possible to add the relation $f(t, t')$ to our logic, where t and t' are members of non-recursive types, although we do not give any details here.

3.1.2 An Informal View of LISA

We now provide intuition for how one formalizes relations over records in our specification language. An example of a record (later modeled as a feature tree) might be:

```
step(status: initial,
     process_1: noncritical,
     next: step(process_1: critical,
                process_2: critical,
                next: stop))
```

The record consists of identifiers (here: step, stop, initial, critical, noncritical) called *labels* and of field selectors (here: *status*, $process_1$, $process_2$, *next*) called *features*. It is a nested record: the value in each record field is itself again a record (possibly without further record fields, *i.e.* a label only). A label does not fix the record fields below it. Records can be graphically represented as trees whose nodes are labeled by labels and whose arcs are labeled by features (see Figure 3.1).

The record above is a solution of the Lisa formula

$$\phi(t) \equiv \quad \forall p.\, \mathsf{critical}(t, p.process_1) \wedge \mathsf{critical}(t, p.process_2) \Rightarrow \mathsf{stop}(t, p.next) \quad (3.1)$$

which expresses: every (sub-) record, where both $process_1$ and $process_2$ are critical, has the value stop in its record field *next*[1]. The record also satisfies a Lisa description of a second kind: it belongs to the defined type *Computation* of all those records that have a record of the same type *Computation* in their record field *next*, or their label is stop. The type could be declared by the following Lisa type definition.

$$Computation = next : Computation \mid \mathsf{stop}$$

In Section 3.4.4, we will see an example of a Lisa formula that combines the two kinds of description; it is precisely this combination of a base logic of feature trees with types that provides us with both a high-level specification language and complexity guarantees.

3.2 The Logic of Lisa

We now introduce the base logic of Lisa. We assume a fixed signature $\langle \mathcal{F}, \mathcal{L} \rangle$ of binary symbols $f \in \mathcal{F}$ called *features* and of binary symbols $l \in \mathcal{L}$ called *labels*. Lisa is a two-sorted first-order logic; we assume an infinite set of feature tree variables $\mathcal{V}_2$ and an infinite set of position variables $\mathcal{V}_1$. We will use X, t, s, ... as meta-variables ranging over $\mathcal{V}_2$ and we will use p, q, ... as meta-variables ranging over $\mathcal{V}_1$. The

[1]Syntax will be described below.

formulae of LISA are generated following the grammar below.

$$\phi ::= f(p,q) \mid l(t,p) \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\,\phi \mid \exists X.\,\phi, \qquad (3.2)$$
$$\text{with } f \in \mathcal{F},\ l \in \mathcal{L},\ p \in \mathcal{V}_1, \text{ and } X \in \mathcal{V}_2\,.$$

Other connectives and quantifiers are of course defined in the standard way.

We reserve the term LISA *formula* to formulae of the LISA logic without free position variables; thus, a LISA formula always defines a relation over feature trees. The atomic formulae have the intuitive meaning.

$l(t,p)$	"the tree t has label l at position p"
$f(p_1,p_2)$	"the position p_2 is the feature f down from the position p_1"

The *interpretation domain* $\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle} = \langle \mathcal{D}_\mathcal{T}, \mathcal{D}_\mathcal{P}\rangle$ consists of the *domain of feature trees* $\mathcal{D}_\mathcal{T} = \{t \mid t\colon \mathcal{D}_\mathcal{P} \to \mathcal{L}\}$ and the *domain of positions* $\mathcal{D}_\mathcal{P} = \mathcal{F}^*$. A feature tree $t \in \mathcal{D}_\mathcal{T}$ consists of nodes in $\mathcal{D}_\mathcal{P}$ with labels in $\mathcal{L}$; we will write $(p,l) \in t$ instead of $t(p) = l$. We also require that the domain of a feature tree t is prefix-closed, *i.e.* $(p.f,l) \in t$ implies $(p,l') \in t$, which amounts to giving a dummy label to "non-labeled" nodes. We may picture a feature tree as a tree with nodes labeled in $\mathcal{L}$ and edges labeled in $\mathcal{F}$; no node has two outgoing edges with the same label.

Feature symbols f are interpreted as binary relations R_f over $\mathcal{D}_\mathcal{P} \times \mathcal{D}_\mathcal{P}$, namely $(p_1,p_2) \in R_f$ iff $p_1.f = p_2$. Labels l are interpreted as binary relations R_l over $\mathcal{D}_\mathcal{T} \times \mathcal{D}_\mathcal{P}$, namely $(t,p) \in R_l$ iff $(p,l) \in t$, *i.e.*the node with the path p is labeled with the symbol l in t. We use, as in (3.1), the following abbreviation: for $1 \leq n$,

$$l(t, p.f_1.f_2.\cdots.f_n) \overset{def}{=} \exists q_1.\cdots.\exists q_n.\, f_1(p,q_1) \wedge \cdots \wedge f_n(q_{n-1},q_n) \wedge l(t,q_n).$$

A LISA *substitution* σ is a pair of mappings $\sigma = (\sigma_1,\sigma_2)$, with $\sigma_1\colon \mathcal{V}_1 \to \mathcal{D}_\mathcal{P}$ and $\sigma_2\colon \mathcal{V}_2 \to \mathcal{D}_\mathcal{T}$, and for $p \in \mathcal{V}_1$, $\sigma(p) = \sigma_1(p)$ and for $t \in \mathcal{V}_2$, $\sigma(t) = \sigma_2(t)$. The satisfiability of a formula ϕ relative to a substitution σ and a domain, denoted by $\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\text{lisa}} \phi$, is defined inductively as follows.

Definition 3.2.1 *Satisfiability for* LISA

$$\begin{array}{lll}
\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} f(p,q), & \textit{if} & (\sigma(p),\sigma(q)) \in R_f \\
\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} l(t,p), & \textit{if} & (\sigma(p),\sigma(t)) \in R_l \\
\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} \neg\phi, & \textit{if} & \mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \not\models_{\textit{lisa}} \phi \\
\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} \phi_1 \vee \phi_2, & \textit{if} & \mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} \phi_1 \textit{ or } \mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} \phi_2 \\
\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} \exists p.\,\phi, & \textit{if} & \mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma[i/p] \models_{\textit{lisa}} \phi, \textit{ for some } i \in \mathcal{D}_\mathcal{P} \\
\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\textit{lisa}} \exists X.\,\phi, & \textit{if} & \mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma[t/X] \models_{\textit{lisa}} \phi, \textit{ for some feature tree } t \in \mathcal{D}_\mathcal{T}
\end{array}$$

If $\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\text{lisa}} \phi$, we say that σ *satisfies*, or is a *model* of, ϕ. We call a formula ϕ *valid*, and we write $\models_{\text{lisa}} \phi$, if for every substitution σ, σ satisfies ϕ.

Note that in Section 2.5.2 we explained that the logic WS2S is not a logic over trees in the direct sense. In contrast, the LISA logic is a logic over trees in the same direct sense that arithmetic is a logic over numbers.

3.3 Compiling the LISA Logic into WS2S

We next describe the compilation of LISA formulae over feature trees into WS2S formulae. Together with the decision procedure for WS2S, this yields, in some sense, the operational semantics of LISA specifications. The idea is very simple: feature trees are encoded by tuples of position sets, one set for each label, with the restriction that each position $p \in \mathcal{D}_{\mathcal{P}}$ occurs in at most one of these sets.

We next define the (effective) bijection $\lceil\cdot\rceil$ between formulae of LISA and WSKS (defined in Section 2.5.1). Let the set of labels be $\mathcal{L} = \{l_1, \ldots, l_n\}$ and the set of features be $\mathcal{F} = \{f_0, \ldots, f_{k-1}\}$. Let ϕ a LISA formula. We assign to each tree variable t of ϕ the n-tuple[2] of the WSKS second-order variables $P_1^t, \ldots, P_n^t$. The variable P_i^t will be used to store the positions in t that are labeled with the label l_i. We assign to each position variable p of ϕ the WSKS first-order variable p. We set

$$\begin{array}{rcll}
\lceil f_j(p,q)\rceil & = & \mathsf{s}_j(p,q) & \text{for } j = 0,\ldots,k-1, \\
\lceil l_i(t,p)\rceil & = & P_i^t(p) \wedge \bigwedge_{j\neq i} \neg P_j^t(p) & \text{for } i = 1,\ldots,n \\
\lceil \phi_1 \vee \phi_2\rceil & = & \lceil\phi_1\rceil \vee \lceil\phi_2\rceil & \\
\lceil \neg\phi\rceil & = & \neg\lceil\phi\rceil & \\
\lceil \exists\, p.\, \phi\rceil & = & \exists p.\, \lceil\phi\rceil & \\
\lceil \exists\, t.\, \phi\rceil & = & \exists P_1^t \ldots P_n^t.\, \lceil\phi\rceil &
\end{array}$$

Note that we associated to each feature f_i the successor s_i, for $0 \leq i < k$. The following theorem expresses the correctness of the compilation of the LISA logic that we have defined above.

Theorem 3.3.1 *The* LISA *formula ϕ is valid over the domain of feature trees if and only if the* WSKS *formula $\lceil\phi\rceil$ is valid over the domain of paths (words in $[k]^*$). Formally,*

$$\models_{lisa} \phi \textit{ iff } \models_{WSkS} \lceil\phi\rceil.$$

[2]In practice, we encode labels using bit patterns over $\lceil log_2(k)\rceil$ second-order variables.

Proof Given a LISA substitution σ, we assign it the WSKS substitution $\lceil\sigma\rceil$ where

$$\begin{aligned}\lceil\sigma\rceil(P_i^t) &= \{p \in \mathcal{D}_{\mathcal{P}} \mid (p, l_i) \in \sigma(t)\},\\ \lceil\sigma\rceil(p) &= \sigma(p).\end{aligned}$$

We can prove by structural induction over LISA formulae ϕ that for all substitutions σ,

$$\mathcal{D}_{\langle\mathcal{F},\mathcal{L}\rangle}, \sigma \models_{\text{lisa}} \phi \text{ iff } [k]^*, \lceil\sigma\rceil \models_{\text{wsks}} \lceil\phi\rceil$$

and, moreover, if $[k]^*, \alpha \models_{\text{wsks}} \lceil\phi\rceil$, *i.e.* the WSKS substitution α is a solution of $\lceil\phi\rceil$, then α is of the form $\alpha = \lceil\sigma\rceil$.

The statement follows directly from the definitions of the mappings $\lceil\cdot\rceil$ for each atomic LISA formula. The induction steps for $\vee$ and $\neg$ are evident; the one for $\exists$ follows from the bijectivity of the mapping between solutions of formulae ϕ of LISA logic and solutions of the corresponding WSKS formulae $\lceil\phi\rceil$. ■

In order to show that LISA is as expressive as WSKS, we need to give the translation from WSKS formulae into LISA formulae. This is a simple embedding. Let the set of labels be $\mathcal{L} = \{d\}$ and the set of features be $\mathcal{F} = \{f_0, \ldots, f_{k-1}\}$. We assign to each second-order value $X \subseteq [k]^*$ the feature tree t_X with $t_X(p) = d$ for all $p \in X$. The WSKS formulae $\mathsf{s}_j(p,q)$ become $f_j(p,q)$, and the formulae $X(p)$ become $d(t_X, p)$.

Corollary 3.3.2 *The* LISA *logic and* WSKS *are equiexpressive.*

3.4 LISA Types

We now build upon the kernel LISA logic by adding a language of types. Let us begin by considering a simple example: binary trees whose labels come from the set $\{a, b, c, d\}$. In a programming language like ML, we might formalize this as:

$$\begin{aligned}&\mathit{datatype\,Tag} &&= a \mid b \mid c \mid d;\\ &\mathit{datatype\,BinTree} &&= \mathit{bin\ of\ }(\mathit{Tag} \times \mathit{BinTree} \times \mathit{BinTree}) \mid \mathit{leaf};\end{aligned}$$

Types specify constraints on the store of the computer; the types above constrain the contents of members *Tag* to have values among the given labels, and members of *BinTree* are trees with a given shape and labeling.

Our type system for LISA formalizes types as systems of recursively defined constraints over feature trees. We formalize the above types as:

$$\begin{aligned}\mathit{Tag} &= \mathsf{a} \vee \mathsf{b} \vee \mathsf{c} \vee \mathsf{d}\\ \mathit{BinTree} &= \mathsf{bin}(\mathit{data}{:}\mathit{Tag},\ \mathit{left}{:}\mathit{BinTree},\ \mathit{right}{:}\mathit{BinTree}) \vee \mathsf{leaf}\end{aligned} \tag{3.3}$$

Types T denote regular sets of trees; hence, they are integrated into the kernel LISA logic as unary predicates over trees. The intended use of types is with relativized quantification. As usual, we write $\forall\, t{:}T.\ \phi$ for $\forall\, t.\ (T(t) \Rightarrow \phi)$ and $\exists\, t{:}T.\ \phi$ for $\exists\, t.\ (T(t) \wedge \phi)$.

Operationally, types can be integrated in a decision procedure for WSKS as follows. A type definition T is compiled to an equivalent deterministic bottom-up tree automaton $\mathcal{A}_T$, as we describe below. The standard decision procedure for WSKS works by processing formulae bottom up, replacing subformulae by tree automata; the procedure can easily be modified such that when encountering the predicate application $T(t)$ in a formula, the automaton $\mathcal{A}_T$ is used. This approach fits, for example, with the already existing library functionality of the MONA system. There, a user can write libraries of predicates $p(t)$ defined in WS2S, and use $p(t)$ as atomic subformulae in subsequent definitions or for theorem statements (*i.e.* WS2S formulas). Each such definition is compiled into an automaton $\mathcal{A}_p$, once and for all, which is used in the decision procedure of the MONA system like a pre-compiled module. The system can call an automaton $\mathcal{A}_T$ stemming from a type definition in exactly the same way as $\mathcal{A}_p$. So, the difference between the two kinds of automata $\mathcal{A}_T$ and $\mathcal{A}_p$ lies in the ways they are specified, not in their use. If the automaton is specified by type definitions, then the compilation has a complexity different from the one of the general WSKS decision procedure. As we will see, it is linear in the practically interesting subcase where all types defined in one type system, which can be seen as forming one library module, denote pairwise disjoint sets.

We now explain the details of this integration. We give the syntax and semantics of types and their translation into tree automata. We analyze the complexity of the translation and the possible size of the resulting automata.

We introduce here some notation needed to define the semantics of the LISA types and their compilation to tree automata.

Let k be a natural number and $L_0, \ldots, L_{k-1}$ be k-ary tree languages. Furthermore, for $a \in \Sigma$ and $L_i \subseteq T_\Sigma$ for $i \in [k]$ we define the *root concatenation* by

$$a \cdot (L_0, \ldots, L_{k-1}) \overset{def}{=} \{a(t_0, \ldots, t_{k-1}) \mid t_i \in L_i \text{ for } i \in [k]\}\,.$$

We observe the simple facts.

$$a \cdot ((L_0, \ldots, L_{k-1}) \cap (M_0, \ldots, M_{k-1})) = a \cdot (L_0, \ldots, L_{k-1}) \cap a \cdot (M_0, \ldots, M_{k-1}) \quad (3.4)$$

$$a \cdot ((L_0, \ldots, L_{k-1}) \cup (M_0, \ldots, M_{k-1})) = a \cdot (L_0, \ldots, L_{k-1}) \cup a \cdot (M_0, \ldots, M_{k-1}) \quad (3.5)$$

$$\overline{(L_0, \ldots, L_{k-1})} = \bigcup_{i \in [k]} (T_\Sigma, \ldots, T_\Sigma, \overline{L_i}, T_\Sigma, \ldots, T_\Sigma) \quad (3.6)$$

$$\overline{a \cdot (L_0, \ldots, L_{k-1})} = a \cdot \overline{(L_0, \ldots, L_{k-1})} \cup \bigcup_{b \in \Sigma \setminus \{a\}} b \cdot (T_\Sigma, \ldots, T_\Sigma) \quad (3.7)$$

Let $\mathbf{X} = \{X_0, \dots, X_{k-1}\}$ be a set of variables ranging over sets of trees in T_Σ and m be a natural number. We define the Boolean algebra $\mathcal{R}^k_m$ as the set of all "language" functions $F : (\mathsf{Pow}(T_\Sigma))^m \to (\mathsf{Pow}(T_\Sigma))^k$, defined in terms of the empty set $\emptyset$, the set T_Σ, the variables in $\mathbf{X}$ and the set operations union $\cup$, intersection $\cap$, and complement $\bar{\ }$. For example, the function: $F(X, Y, Z) = \overline{X \cup Y} \cap Z$ is an element from $\mathcal{R}^1_3$. A function $F \in \mathcal{R}^k_m$ is said *complement free*, if it does not involve the complement operation. Let $\mathcal{R}^m$ be the set of functions $F : (\mathsf{Pow}(T_\Sigma))^m \to \mathsf{Pow}(T_\Sigma)$, defined in terms of subsets of $\mathsf{Pow}(T_\Sigma)$, of functions in $\mathcal{R}^k_m$, the root concatenation, and the set operations $\cap$, $\cup$, and $\bar{\ }$. Analogously, a function in $\mathcal{R}^m$ is complement free, if it does not involve the complement operation and complement free functions from $\mathcal{R}^k_m$.

Let $\mathbf{x} = \{x_0, \dots, x_{k-1}\}$ be a set of Boolean variables. We define the set $\mathcal{B}^k_m$ of the Boolean functions $f : \mathbb{B}^m \to \mathbb{B}^k$ defined in terms of the truth values false and true, the variables in $\mathbf{x}$ and the connectives $\wedge$, $\vee$ and $\neg$. The set $\mathcal{B}^k_m$ together with the Boolean connectives also forms a Boolean algebra that is obviously isomorphic to the algebra $\mathcal{R}^k_m$. For the remainder of this chapter, if F is in $\mathcal{R}^k_m$, then f denotes its isomorphic image in $\mathcal{B}^k_m$: f is obtained from F by replacing $\emptyset$ by false, T_Σ by true, each variable X_i by the variable x_i, the intersection $\cap$ by conjunction $\wedge$, the union $\cup$ by disjunction $\vee$ and the complement $\bar{\ }$ by negation $\neg$.

Definition 3.4.1 *Let $F_1, \dots, F_m$ be complement free functions from $\mathcal{R}^m$. The function $\Phi : (\mathsf{Pow}(T_\Sigma))^m \to (\mathsf{Pow}(T_\Sigma))^m$ defined by $\Phi(D_1, \dots, D_m) = (D'_1, \dots, D'_m)$, where $D'_i = F_i(D_1, \dots, D_m)$. The function Φ is called* fixed-point operator.

The domain $(\mathsf{Pow}(T_\Sigma))^m$, ordered pointwise under the subset inclusion, forms a *complete partial order (cpo)* where $(\emptyset, \dots, \emptyset)$ acts as the least element and $(T_\Sigma, \dots, T_\Sigma)$ acts as the greatest element. The fixed-point operator Φ of Definition 3.4.1 is *monotonous*; that is

$$\text{if } (D_1, \dots, D_m) \subseteq (D'_1, \dots, D'_m) \text{ then } \Phi(D_1, \dots, D_m) \subseteq \Phi(D'_1, \dots, D'_m)\,.$$

This can be easily proven taking into consideration that the functions F_i are complement free. By the Knastar-Tarski fixed-point theorem, the function Φ has a least fixed-point. We denote with $\mathsf{fix}(\Phi)$ the least fixed-point of Φ and with $\mathsf{fix}^i(\Phi)$ its i-th component.

3.4.1 Syntax and Semantics

We assume given a finite set $\mathcal{L}$ of labels and a finite set $\mathcal{F}$ of k features; we set $\mathcal{F} = \{f_0, \dots, f_{k-1}\}$. A *type system* $\mathcal{T}$ is a conjunction of *type equations*,

$$\mathcal{T} \equiv \{T_1 = \theta_1, \dots, T_m = \theta_m\}\,,$$

between pairwise different *declared types* T_j and *type bodies* θ_j. The syntax of the bodies θ_j is given by the grammar

$$\theta ::= \ a(f_0{:}S_0, \ldots, f_{k-1}{:}S_{k-1}) \mid f{:}S \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2 \, , \tag{3.8}$$

where $a \in \mathcal{L}$ and $f \in \mathcal{F}$, and S, S_0, $\ldots$, S_{k-1} are declared types[3]. In addition, we assume given a type Λ whose interpretation will be defined later as the singleton set consisting of the empty tree λ, and also we assume given a type $\top$ whose interpretation will be defined later as the set T_Σ.

The syntax of LISA formulae is extended with quantification relativized to types. The meaning of type membership "$t \in T$" for a feature tree t is intuitively clear.

(1) If T is Λ then t is the empty tree, $t = \lambda$.

(2) If T is the type $\top$ then t is an arbitrary tree from T_Σ.

(3) If T is defined by $a(f_0{:}S_0, \ldots, f_{k-1}{:}S_{k-1})$, then t is labeled with a at the root and it must have a subtree t_i of type S_i at subtree position f_i, for $0 \leq i < k$, and can have arbitrary trees at the other subtree positions.

(4) If T is defined by $f{:}S$, then t is a tree that has at the position f a tree of type S.

Formally, the meaning of a type T is a set of feature trees that we denote with $[\![T]\!]$. The meaning of Λ is $[\![\Lambda]\!] = \{\lambda\}$ and the meaning of $\top$ is $[\![\top]\!] = T_\Sigma$. Let $\mathcal{T}$ be the type system $\mathcal{T} \equiv \{T_1 = \theta_1, \ldots, T_m = \theta_m\}$. We associate with each type T in $\mathcal{T}$ a variable X_T and introduce the following mapping ϱ defined on LISA types by:

$$\varrho(T) = \begin{cases} X_T, & \text{if } T \in \mathcal{T} \\ \{\lambda\}, & \text{if } T = \Lambda \\ T_\Sigma, & \text{if } T = \top \end{cases}$$

Let τ be the following translation:

$$\tau(\theta) = \begin{cases} a \cdot (\varrho(S_0), \ldots, \varrho(S_{k-1})), & \text{if } \theta = (f_0{:}S_0, \ldots, f_{k-1}{:}S_{k-1}) \\ \bigcup_{a \in \mathcal{L}} a \cdot (\top, \ldots, \top, \varrho(S), \top \ldots, \top), & \text{if } \theta = f{:}S \\ \tau(\theta_1) \cup \tau(\theta_2), & \text{if } \theta = \theta_1 \vee \theta_2 \\ \tau(\theta_1) \cap \tau(\theta_2), & \text{if } \theta = \theta_1 \wedge \theta_2 \end{cases}$$

[3]We will use T, S, S_1, $S_2 \ldots$ as metavariables ranging over the types T_1, $\ldots$, T_m, Λ, and $\top$.

Let F_i be the complement free function defined by $F_i(X_0, \ldots, X_m) = \tau(\theta_i)$ for $1 \leq i \leq m$ and let $\Phi_{\mathcal{T}}$ be the fixed-point operator defined means the above functions F_i's. We define the semantics of the type system $\mathcal{T}$ as the least fixed-point of $\Phi_{\mathcal{T}}$ and we write $[\![\mathcal{T}]\!] = \mathsf{fix}(\Phi_{\mathcal{T}})$. Furthermore, we define the semantics of T_i as the i-th component of this least fixed-point, that is $[\![T_i]\!] = \mathsf{fix}^i(\Phi_{\mathcal{T}})$.

Example 3.4.1 Consider the example above defining the binary-tree type. We are given the set of labels $\mathcal{L} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{bin}, \mathsf{leaf}\}$ and the set of features $\mathcal{F} = \{\mathit{data}, \mathit{left}, \mathit{right}\}$. In order to shorten notation for trees, we write a for the tree $\mathsf{a}(\lambda, \lambda, \lambda)$, leaf for the tree $\mathsf{leaf}(\lambda, \lambda, \lambda)$, etc. We implicitly identified the features *data, left, right* with the first, second and third successor respectively. The meaning of the types *Tag* and *BinTree* are as follows.

$$\begin{array}{lcl} [\![\mathit{Tag}]\!] & = & \{\mathsf{a},\ \mathsf{b},\ \mathsf{c},\ \mathsf{d}\} \text{ and} \\ [\![\mathit{BinTree}]\!] & = & \{\mathsf{leaf},\ \mathsf{bin}(\mathsf{a}, \mathsf{leaf}, \mathsf{leaf}), \ldots\} \end{array}$$

□

3.4.2 Compiling the LISA Types into Tree Automata

Given the type system $\mathcal{T}$ declaring the types $T_1, \ldots, T_m$, we show how to construct a family $\mathcal{A}_{\mathcal{T}} = (\mathcal{A}_i)_{1 \leq i \leq m}$ of top-down alternating tree automata, such that each automaton $\mathcal{A}_i$ accepts exactly the trees in $[\![T_i]\!]$. The automata will be defined so that they only differ in their starting states.

In a preliminary step, we define systems of language equations and show that they are equivalent to ATTA's (see Definition 2.2.5). After that, we show how LISA types can be translated into systems of language equations.

3.4.3 Systems of Language Equations

Systems of language equations of different forms have been studied by a number of authors [BL80, Lei81a, Sal69]. Brzozowski and Leiss in [BL80], in particular were interested to connecting systems of language equations for regular word languages with sequential networks and alternating word automata (Boolean automata in their terminology).

In the following, we generalize the definition of Brzozowski and Leiss [BL80] from the word case to the tree case and we illustrate their relationship to alternating top-down tree automata.

Definition 3.4.2 *A system $\mathcal{S}$ of language equations is a conjunction of equations,*

$$\mathcal{S} \equiv \{X_1 = \theta_1, \ldots, X_m = \theta_m\},$$

between different declared variables X_i and bodies θ_i. The syntax of the bodies is given by following grammar

$$\theta_i ::= \bigcup_{a \in \Sigma} a \cdot F_a^i(X_1, \ldots, X_m) \cup \zeta_i \,,$$

where $F_a^i \in \mathcal{R}_m^k$ and ζ_i is either the empty set $\emptyset$ or the singleton $\{\lambda\}$[4].

To the system $\mathcal{S}$ we associate the fixed-point operator

$$\Phi_{\mathcal{S}} : (\mathsf{Pow}(T_\Sigma))^m \to (\mathsf{Pow}(T_\Sigma))^m$$

defined by means of the functions

$$F_i(X_1, \ldots, X_m) = \bigcup_{a \in \Sigma} a \cdot F_a^i(X_1, \ldots, X_m) \cup \zeta_i, \text{ for } 1 \leq i \leq m$$

as specified in Definition 3.4.1.

The semantics of $\mathcal{S}$, $L(\mathcal{S})$, is an m-tuple of languages. We distinguish two cases. First, if the functions F_a^i for $1 \leq i \leq m$ are complement free, then the semantics of $\mathcal{S}$ is simply the least fixed-point $L(\mathcal{S}) = \mathsf{fix}(\Phi_{\mathcal{S}})$ of $\Phi_{\mathcal{S}}$. In the second case, the functions F_a^i are arbitrary and we eliminate the complement operation from the equations of $\mathcal{S}$ by computing a new system of language equations $\mathcal{S}'$, and then we define the semantics of $\mathcal{S}$ using the semantics of $\mathcal{S}'$. We proceed as follows. First, we duplicate the equations of $\mathcal{S}$, by introducing for each equation $X_i = \theta_i$ the new equation $\overline{X_i} = \overline{\theta_i}$ for $1 \leq i \leq m$. Next, we use the equations (3.4), (3.5), (3.6), and (3.7) to transform each of the expressions θ_i and $\overline{\theta_i}$ into their negation normalforms and replace after that each negated variable $\overline{X_i}$ with a new variable Y_i. Thus, we obtained a new system $\mathcal{S}'$ of language equations over the variables $X_1, \ldots, X_m, Y_1, \ldots, Y_m$ where the body definitions are complement free. The semantics of $\mathcal{S}$ is defined as the first m sets of the fixed-point of $\Phi_{\mathcal{S}'}$, namely $L(\mathcal{S}) = (\mathsf{fix}^1(\Phi_{\mathcal{S}'}), \ldots, \mathsf{fix}^m(\Phi_{\mathcal{S}'}))$. In the following we will use $L_{\mathcal{S}}(X_i)$ for the i-th component of $L(\mathcal{S})$.

The fixed-point approach defines the semantics of the system of language equations, but does not specify a practical algorithm to compute the solution. Below, we show how the solution of such systems can be represented by a family of automata. For completeness, we describe how for the word case the fixed-point is traditionally calculated.

For the word case ($k = 1$) it has been shown (*cf.* [Lei81a, BL80]) that the solution $L(\mathcal{S})$ is unique and regular. The key idea for this fact is based on the following so called *Arden's Lemma* [Ard61]: The equation

$$X = U\,X \cup V, \text{ with } U, V \subseteq \Sigma^* \text{ and } \epsilon \notin U,$$

[4]In the case where $k = 1$, we have $\{\epsilon\}$ instead of $\{\lambda\}$.

has the unique solution $X = U^*V$, which is regular, if both U and V also are regular. The idea behind solving a system of language equations $\mathcal{S}$ is to apply Arden's lemma using the following rules:

For $X = \bigcup_{a\in\Sigma} a \cdot M_a \cup \zeta_X$, and $Y = \bigcup_{a\in\Sigma} a \cdot N_a \cup \zeta_Y$, where $M_a, N_a \subseteq \Sigma^*$,

(1) $X \cup X = \bigcup_{a\in\Sigma} a \cdot (M_a \cup N_a) \cup (\zeta_X \cup \zeta_Y)$

(2) $X \cap X = \bigcup_{a\in\Sigma} a \cdot (M_a \cap N_a) \cup (\zeta_X \cap \zeta_Y)$

(3) $\overline{X} = \bigcup_{a\in\Sigma} a \cdot \overline{M_a} \cup (\{\epsilon\} \setminus \zeta_X)$

Example 3.4.2 Let $\mathcal{S}$ be a language system defined as follows:

$$\begin{aligned} X_1 &= a \cdot X_2 \cup \{\lambda\} \\ X_2 &= a \cdot (X_1 \cup X_2) \cup b.X_1 \\ X_3 &= b \cdot \overline{X_1} \end{aligned}$$

We can apply Arden's lemma to the second equation and we obtain the new equation $X_2 = a^*(a+b) \cdot X_1$. With this in hand, we apply again Arden's lemma to the first equation and obtain the solution $(a^+(a+b))^*$ of X_1. The solutions of X_2 and X_3 are calculated similarly. We thus obtain

$$\begin{aligned} X_1 &= (a^+(a+b))^* \\ X_2 &= a^*(a+b)(a^+(a+b))^* \\ X_3 &= b\overline{(a^+(a+b))^*} \end{aligned}$$

□

Now, let us turn our attention to the automata representation of solutions of systems of language equations.

Lemma 3.4.3 *For every system $\mathcal{S}$ of language equations over the variables $X_1, \ldots, X_m$ there is a family of alternating top-down tree automata $(\mathcal{A}_i)_{i\leq m}$ such that $L_{\mathcal{S}}(X_i) = L(\mathcal{A}_i)$.*

Proof We define the family of alternating automata $\mathcal{A}_i = (Q, x_i, \delta, F)$, for $1 \leq i \leq m$ as follows: for each variable X_j we associate a state x_j, $Q = \{x_1, \ldots, x_m\}$, the initial Boolean state of the automaton $\mathcal{A}_i$ is the state x_i, and the final states set F contains all states x_j for which $\zeta_j = \{\lambda\}$. The transition function $\delta : Q \times \Sigma \rightarrow B(Q \times [k])$ is defined by: $\delta(x_j, a) = f_a^j$, where $X_j = \bigcup_{a\in\Sigma} a \cdot F_a^j(X_1, \ldots, X_m) \cup \zeta_j$ and $f_a^j \in \mathcal{B}_m^k$ is the isomorphic image of F_a^j, for $1 \leq j \leq m$. Following this construction, $L(\mathcal{A}_i) = L_{\mathcal{S}}(X_i)$ holds. ■

Example 3.4.3 The family of automata associated to the system $\mathcal{S}$ from Example 3.4.2 is given by:

$$\mathcal{A}_i = (\{x_1, x_2, x_3\}, x_i, \delta, \{x_1\}), \text{ for } 1 \leq i \leq 3$$

The transition function δ is given by:

$$\begin{aligned} \delta(x_1, a) &= x_2, & \delta(x_1, b) &= \mathsf{false}, \\ \delta(x_2, a) &= x_1 \vee x_2, & \delta(x_2, b) &= x_1, \\ \delta(x_3, a) &= \mathsf{false}, & \delta(x_3, b) &= \neg x_1 \,. \end{aligned}$$

□

Now, we show that an alternating top-down automaton can be represented by a system of language equations.

Lemma 3.4.4 *For every* ATTA $\mathcal{A} = (Q, I, \delta, F)$ *with* $Q = \{x_1, \ldots, x_m\}$ *there is a system* $\mathcal{S}$ *of language equations over the variables* $X_1, \ldots, X_m$ *such that* $L(\mathcal{A}_i) = L_{\mathcal{S}}(X_i)$*, where* $\mathcal{A}_i = (Q, x_i, \delta, F)$.

Proof $\mathcal{S}$ is defined by the equations $X_i = \bigcup_{a \in \Sigma} a \cdot F_a^i(X_1, \ldots, X_m) \cup \zeta_i$, where ζ_i is $\emptyset$ if $\lambda \notin L(\mathcal{A}_i)$ and $\{\lambda\}$ otherwise. The function F_a^i is the isomorphic image of the Boolean function $\delta(x_i, a)$. ■

Example 3.4.4 Consider the alternating top-tree automaton of Example 2.2.2. By applying the construction of the previous proof we obtain the following system $\mathcal{S}$ of language equations

$$\begin{aligned} X_0 = &\ f \cdot ((X_1, X_2) \cup (X_2, X_1)) \\ X_1 = &\ a \cdot (X_3, X_3) \\ X_2 = &\ b \cdot (X_3, X_3) \\ X_3 = &\ \{\lambda\} \end{aligned}$$

where the solution $L(\mathcal{S})$ is as expected. Recall that we suppress the empty subtrees.

$$\begin{aligned} L_{\mathcal{S}}(X_0) &= \{f(a, b), f(b, a)\}, & L_{\mathcal{S}}(X_1) &= \{a\} \\ L_{\mathcal{S}}(X_2) &= \{b\}, \text{ and} & L_{\mathcal{S}}(X_3) &= \{\lambda\} \end{aligned}$$

□

Lemma 3.4.5 *Every* LISA *type system* $\mathcal{T}$ *can be transformed into a system of language equations* $\mathcal{S}$ *such that for each type* T, $[\![T]\!] = L_{\mathcal{S}}(T)$ *holds.*

Proof For the types Λ and $\top$, we introduce the equations

$$\begin{aligned} X_\Lambda &= \bigcup_{a\in\mathcal{L}} a\cdot(\emptyset,\ldots,\emptyset) \cup \{\lambda\} \\ X_\top &= \bigcup_{a\in\mathcal{L}} a\cdot(X_\top,\ldots,X_\top) \cup \{\lambda\} \end{aligned}$$

For each type $T \in \mathcal{T}\cup\{\Lambda,\top\}$ we associate the variable X_T. For a type declaration $T=\theta$ in $\mathcal{T}$, we have the equation $X_T=\tau(\theta)$ in $\mathcal{S}$ and where the translation τ is given below.

$$\tau(\theta) = \begin{cases} a\cdot(X_{S_0},\ldots,X_{S_{k-1}})\cup\emptyset, & \text{if } \theta=(f_0{:}S_0,\ldots,f_{k-1}{:}S_{k-1}) \\ \bigcup_{a\in\mathcal{L}} a\cdot(X_\top,\ldots,X_\top,X_S,X_\top,\ldots,X_\top)\cup\emptyset & \text{if } \theta=f{:}S \\ \tau(\theta_1)\ \mathsf{Or}\ \tau(\theta_2), & \text{if } \theta=\theta_1\vee\theta_2 \\ \tau(\theta_1)\ \mathsf{And}\ \tau(\theta_2), & \text{if } \theta=\theta_1\wedge\theta_2 \end{cases}$$

The operators Or and And are defined by

$$\begin{aligned} &(\textstyle\bigcup_{a\in\Sigma} a\cdot(S_0^a,\ldots,S_{k-1}^a)\cup\zeta_1)\ \mathsf{Or}\ (\bigcup_{a\in\Sigma} a\cdot(R_0^a,\ldots,R_{k-1}^a)\cup\zeta_1) = \\ &\qquad\qquad \textstyle\bigcup_{a\in\Sigma} a\cdot((S_0^a,\ldots,S_{k-1}^a)\cup(R_0^a,\ldots,R_{k-1}^a))\cup\zeta_1\cup\zeta_2) \\ &(\textstyle\bigcup_{a\in\Sigma} a\cdot(S_0^a,\ldots,S_{k-1}^a)\cup\zeta_1)\ \mathsf{And}\ (\bigcup_{a\in\Sigma} a\cdot(R_0^a,\ldots,R_{k-1}^a)\cup\zeta_1) = \\ &\qquad\qquad \textstyle\bigcup_{a\in\Sigma} a\cdot((S_0^a,\ldots,S_{k-1}^a)\cap(R_0^a,\ldots,R_{k-1}^a))\cup\zeta_1\cap\zeta_2) \end{aligned}$$

■

Theorem 3.4.6 *Every* LISA *types system T_1, …, T_m can be transformed into a family of bottom-up tree automata $(\mathcal{A}_i)_{i=1}^m$ such that each automaton $\mathcal{A}_i$ has at most $2^{O(m)}$ states and $L(\mathcal{A}_i)=[\![T_i]\!]$.*

Proof It follows from Theorem 2.2.6, Lemma 3.4.3, and Lemma 3.4.5. ■

Corollary 3.4.7 *The* LISA *type system is as expressive as alternating top-down tree automata and therefore also as expressive as the* LISA *kernel logic itself.*

The Size of the Automaton

For a type system $\mathcal{T}$ with n equations our construction yields a deterministic bottom-up tree automaton of size $2^{O(n)}$. This is the upper bound for the general case of LISA types. One can now look for natural semantic restrictions on type definitions with better bounds. We will define a semantic property that is sufficient to guarantee that there exists a tree automaton recognizing the declared types in which the number of states is linear in the number of types. This property encompasses a large and natural class of types, including those given in this chapter and those typically encountered in type declarations.

Let $\mathcal{T}$ be a system of nonempty types and $k = |\mathcal{F}|$. We say $\mathcal{T}$ is *disjoint* if all its types are pairwise disjoint, *i.e.* $[\![T]\!] \cap [\![T']\!] = \emptyset$, for all distinct T and T' in $\mathcal{T}$. For any such type system the following holds:

Lemma 3.4.8 *If $\mathcal{T}$ is disjoint, then each type $T \in \mathcal{T}$ has a body that can be transformed into a finite disjunction of formulae θ of the form $a(f_0{:}S_1, \ldots, f_{k-1}{:}S_{k-1})$, where S_i is Λ, $\top$, or a type in $\mathcal{T}$ for $1 \leq i \leq n$.*

We omit the straightforward but tedious proof of this lemma. A consequence of Lemma 3.4.8, is the following:

Lemma 3.4.9 *If $\mathcal{T}$ is disjoint, then there is a deterministic bottom-up k-ary tree automaton $\mathcal{A}_\mathcal{T}$ that recognizes $\mathcal{T}$ and it has at most $|\mathcal{T}|$ states.*

A deterministic bottom-up k-ary tree automaton $\mathcal{A}$ can be transformed into a deterministic bottom-up binary tree automaton $\mathcal{A}'$ whose number of states is linear in the number of the states of $\mathcal{A}$. Hence, for a disjoint type system $\mathcal{T}$ we can build a deterministic bottom-up binary tree automaton $\mathcal{A}$ that recognizes $\mathcal{T}$ and whose number of states is linear in the size of $\mathcal{T}$.

Although disjointness is semantically defined, we define below a sufficient syntactic characterization for disjointness that can be checked in linear time.

Definition 3.4.10 *Let T be a type with body θ. We define $root(T)$ as the set of all labels that can label the root of an element of T. For $T = \theta$, $root(T) = root(\theta)$ and $root(\theta)$ is defined by:*

$$root(\theta) = \begin{cases} \{a\}, & \text{if } \theta = a(f_0{:}S_1, \ldots, f_{k-1}{:}S_{k-1}) \\ \mathcal{L}, & \text{if } \theta = f{:}S \\ root(\theta_1) \cup root(\theta_2), & \text{if } \theta = \theta_1 \vee \theta_2 \\ root(\theta_1) \cap root(\theta_2), & \text{if } \theta = \theta_1 \wedge \theta_2 \end{cases}$$

Lemma 3.4.11 *If the types in $\mathcal{T}$ have disjoint roots then $\mathcal{T}$ is disjoint.*

Example 3.4.5 (continued) Consider the type system $\mathcal{T}$ containing the types *BinTree* and *Tag* of Example 3.4.1. The root of *Tag* is the set $root(Tag) = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and the root of the type *BinTree* is the set $root(BinTree) = \{\mathsf{bin}, \mathsf{leaf}\}$. The roots of both types are disjoint, it follows that $\mathcal{T}$ is disjoint (this fact is obvious for this example). In $\mathcal{T}$ we have three features *tag, left* and *right* that we associate with the projections on the first, second and third component respectively. The deterministic 3-ary bottom-up tree automaton $\mathcal{A} = (A_q)_{q \in \{BinTree,\, Tag\}}$ over the signature $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{bin}, \mathsf{leaf}\}$ that recognizes $\mathcal{T}$ is defined by $\mathcal{A}_q = (\{\Lambda,\ BinTree,\ Tag\},\ \Lambda,\ \delta,\ \{q\})$ with

$$\delta(x,\ \Lambda,\ \Lambda,\ \Lambda) = Tag, \text{ for } x \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$$

$$\delta(\mathsf{leaf},\ \Lambda,\ \Lambda,\ \Lambda) = BinTree$$

$$\delta(\mathsf{bin},\ Tag,\ BinTree,\ BinTree) = BinTree\,.$$

□

3.4.4 An Example

We now illustrate some of the features of LISA with an example taken from [BA90] and which is also considered by Klarlund and Schwartzbach [KNS96]: the correctness of the following toy mutual exclusion algorithm.

```
Turn: Integer range 1..2 := 1; Proc(i))  =
  loop
    a: Non_critical_section_i
    b: Loop exit when Turn = i; end loop;
    c: Critical_section_i;
    d: Turn := i + 1 mod 2
  end Loop;

Proc(0) || Proc(1)
```

This algorithm consists of two processes that execute the program, whose lines are numbered `a` through `d`. The variable `Turn` resides in shared memory. We begin our specification by declaring the following datatypes[5].

```
Turn  = 1 | 2;
Pc    = a | b | c | d;
State = state(pc1:Pc, pc2:Pc, turn:Turn);
Comp  = node(val: State, next:Comp) | done;
```

The type `Comp` formalizes trees that represent sequences of states. Each state has features pointing to the program counters (each storing a program line) and the value of the `Turn` variable.

Not all elements of `Comp` represent valid executions. Hence, we define LISA predicates[6] that further constrain the members of `Comp`.

[5]We use for LISA the following ASCII syntax: `|` for $\vee$, `&` for $\wedge$, and `=>` for $\rightarrow$.

[6]A predicate (defined with `pred` in LISA) is equivalent to (*i.e.* is syntactically interchangeable with) a formula in one or several free variables (tree variables in the case of LISA).

The predicate `Init` describes the start state of both processes.

```
pred  Init(X:State)  = a(X,pc1) & a(X,pc2) & 1(X,turn);
```

The state `X` satisfies `Init` if (1) it is a tree of type `State` and (2) its subtrees at the positions `pc1` and `pc2` are labeled with the program line `a` and the subtree at the position `turn` is labeled with the turn value `1`.

For the transition function, we first declare how a process can execute. The predicate `Step1` is a large conjunction that describes the possible transitions.

```
pred Step1(X:State, Y:State) =
   (a(X,pc1) => b(Y,pc1) & X.turn = Y.turn) &
   (b(X,pc1) => ((1(X,turn)=>c(Y,pc1))&
                 (2(X,turn)=>b(Y,pc1))&(X.turn = Y.turn)))&
   (c(X,pc1) => d(Y,pc1) & X.turn = Y.turn) &
   (d(X,pc1) => a(Y,pc1) & 2(Y,turn)& X.pc2 = Y.pc2);
```

For example, the first conjunct `(a(X,pc1) => b(Y,pc1) & X.turn = Y.turn)` states that the first process can advance from state `X` where `pc1` is at line `a`, to state `Y` where `pc1` is at line `b`, and the value of the `turn` variable remains unchanged. The predicate `Step2` is declared similarly.

```
pred Step2(X:State, Y:State) =
   (a(X,pc2) => b(Y,pc2) & X.turn = Y.turn) &
   (b(X,pc2) => ((1(X,turn)=>b(Y,pc2))&
                  (2(X,turn)=>c(Y,pc2))&(X.turn = Y.turn)))&
   (c(X,pc2) => d(Y,pc2) & X.turn = Y.turn) &
   (d(X,pc2) => a(Y,pc2) & 1(Y,turn) & X.pc1 = Y.pc1);
```

A transition of the system is a step by either process (*i.e.* we assume an interleaving semantics).

```
pred Trans(X:State, Y:State) = Step1(X,Y) | Step2(X,Y) ;
```

Finally, a computation is `Valid` when the first state satisfies `Init` and all pairs stand in the transition relation.[7]

```
pred Valid(X:Comp) = Init(X.val) &
  all p. node(X,p) & node(X,next.p) => Trans(X.p.val,X.p.next.val);
```

Given these definitions, we now define what mutual exclusion means: no two processes are simultaneously in their critical section, line `c`.

```
pred Mutex(X:Comp) =  all p. ~ (c(X,p.val.pc1) & c(X,p.val.pc2));
```

With this, we can formalize the question of whether all valid computations have the mutual exclusion property.

[7]`X.f` is the tree such that `a(X.f,p)` iff `a(X,p.f)` for all labels `a` and for all positions `p`.

```
  all  X:Comp. Valid(X) => Mutex(X);
```

We have a prototype implementation for LISA which is integrated with MONA. For this example, LISA translates the defined predicates and types and produces about 2 pages of formulae in WS2S. These are input to MONA, which takes 2 seconds to process the result, and to report

```
ANALYSIS: Formula is valid
```

thereby verifying that the program does indeed enforce mutual exclusion.

This example illustrates how types and LISA formulae interact. There is a natural decomposition of specifications into types, which express simple properties about the shapes and values of data, and predicates (*e.g.* `Mutex`), which express more complicated constraints. Again, recall that both are equally expressive; the tradeoff is one between conciseness and complexity of the translation. We can also express predicates like `Valid` and `Mutex` as types, but the results would be rather cumbersome (a blow-up in size traded in for a better complexity bound with respect to the larger types) and not very natural.

3.5 Extension of the LISA Type System

We extend the LISA type system by allowing the use of the complements of types. The complement of a type T is syntactically denoted by $\overline{T}$ and its meaning $[\![\overline{T}]\!]$ is, as expected, the set $T_\Sigma \setminus [\![T]\!]$. In the grammar (3.8) we now allow that the metavariables S, S_0, …, S_{k-1} range over the types of $\mathcal{T}$ including Λ and $\top$ as well as their complements. For example, we can declare a type T by $T = f{:}\overline{S}$. The type T intuitively denotes the set of all trees t in T_Σ such that the subtree of t at position f does not belong to the set denoting the type S. Generally, a LISA type system $\mathcal{T}$ specified in the new extended syntax can be translated into a system of language equations $\mathcal{S}$ in the same way as we described in Lemma 3.4.5 and the semantics of $\mathcal{T}$ is also defined as the least fixed-point of the operator $\Phi_{\mathcal{S}}$, $\mathsf{fix}(\Phi_{\mathcal{S}})$.

Our extension of the LISA type system does not increase the expressive power of the LISA type system, but make the type system more concise. Furthermore, as we have seen in Section 3.4.3, we can transform a type system $\mathcal{T}$ involving the complement operation into a type system without the complement operation in a time linear in the number of the original types. Thus, the deterministic bottom-up tree automaton corresponding to a LISA type system with m types has at most 2^{2m} states.

3.6 The Linear Lisa

The reader may have observed that in many cases a Lisa program can be translated into a WS1S formula instead of a WS2S formula. These cases do not only include the case where the set of features is a singleton ($k = 1$), but also other interesting cases where $k > 1$. These cases arise always when the feature trees of the declared types are degenerated trees and can be represented by words. For example, if we reconsider Example 3.4.4 we will notice that the feature `next` is the unique feature that occurs "recursively" and thus the feature trees of the type `Comp` can be encoded as words.

It is also worthwhile to notice that, although the decision procedures for both WS1S and WS2S are non-elementary, practice shows that it is efficient to use the decision procedure of WS1S rather than the decision procedure for WS2S for problems that can be encoded in both logics. This is not surprising, because the algorithms used for deciding WS1S compared to those used for deciding WS2S are simpler, more widespread and optimized.

We have equipped the prototype implementation of Lisa with a mode that we call *Linear Lisa* (LLisa). LLisa provides a type system that is more appropriate for the word case and which is defined on the top of the Lisa type system given in the grammar (3.8) using a lot of syntactic sugar.

LLisa has two primitive types bool and nat and allows the user to define structured types. Let $\mathcal{L}$ be a set of labels and $\mathcal{F}$ be a set of features. A user type in LLisa can be defined using the following schema:

$$\mathsf{data}\, T \;=\; \theta,$$

where data is the key word that precedes every type declaration and θ is the body of the type and has the form

$$\theta ::= T \;\mathsf{union}\; T \mid \mathit{enumerate} \mid \mathit{interval} \mid \mathit{record} \mid \mathit{array},$$

where

$$\begin{array}{lcl} \mathit{enumerate} & ::= & a_1, \ldots, a_n \\ \mathit{interval} & ::= & n_1 \ldots n_2 \\ \mathit{record} & ::= & \mathsf{record}\; \{f_1{:}S_1, \ldots, f_n{:}S_n\} \\ \mathit{array} & ::= & \mathsf{array}\; S_1 \;\mathsf{of}\; S_2 \end{array}$$

where $n_1, n_2 \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathcal{L}$, $f_1, \ldots, f_n \in \mathcal{F}$, and $S_1, \ldots, S_n$ are either declared types or primitive types. In the record type declaration, the types $S_1, \ldots, S_n$ are called component types and the features $f_1, \ldots, f_n$ are also called selectors. In the array declaration, the type S_1 is called index type and S_2 is called element type.

In LLisa, we distinguish two kind of types. *Finite* types which are types that denote finite sets, and *infinite* types which denote infinite sets. The type bool denotes the set of Boolean values and is finite. The type nat stands for the set of natural numbers and is infinite. The union type constructor is only allowed for two finite types; so the union of two types is a finite type. An enumeration type is given by a finite set of labels and thus is finite. An interval is a segment of $\mathbb{N}$ starting from the natural number n_1 and ending with the natural number n_2. Interval types are finite. A record type is finite if all its component types are finite, and is infinite only if at least one of its component types is infinite. For an array type the index type should be either nat or an interval and furthermore it is not allowed that the index and element types are infinite.

The support of user definable types in LLisa can be seen as one step towards providing a specification language equipped with high-level notation. Another step in this direction is to provide high-level language constructors. We describe here the most interesting constructors incorporated in LLisa.

Field Selection Let r be a term whose type T is a record and f be a feature/selector occurring in the declaration of T. We use the notation $r.f$ to access to the component stored in r under the selector f.

Updates of Fields Let s and s' be two variables of the same record type. We write $s' = s\{p_1 = v_1, \ldots, p_n = v_n\}$ to state that the record s' is obtained from s by overwriting the components at the positions $p_1, \ldots, p_n$ with new values $v_1, \ldots, v_n$ respectively.

Conditionals We have three kinds of conditionals. Besides the commonly used constructors if_then and if_then_else, we have a straightforward generalization of these constructors defined by

$$\mathsf{cond}\ c_1 \Rightarrow a_1;\ \ldots;\ c_n \Rightarrow a_n;\ \mathsf{dnoc},$$

where $c_1, \ldots, c_n$ and $a_1, \ldots, a_n$ are formulae. Intuitively, the c_is are conditions and the a_is are actions. The ordering of the actions does matter; that is, the action a_i takes place only if the condition c_i holds and the conditions $c_1, \ldots, c_{i-1}$ do not hold. This construct is syntactic sugar for the formula

$$\bigvee_{1 \le i \le n} \neg c_1 \wedge \ldots \wedge \neg c_{i-1} \wedge c_i \wedge a_i$$

Case Analysis The case analysis expression offers a kind of pattern-matching.

$$\mathsf{case}\ t\ \mathsf{of}\ t_1 \Rightarrow a_1;\ \ldots;\ t_n \Rightarrow a_n;\ \mathsf{esac}$$

The term t has as type T an enumeration or interval. The terms $t_1, \ldots, t_n$ are constants of type T. The a_i's are formulae. The case analysis is a readable form of the following formula.

$$\bigvee_{1 \le i \le n} t = t_i \wedge a_i$$

Example 3.6.1 (Formalizing an Elevator in Linear Lisa)
Our goal here is to demonstrate the high-level notation available in LLisa. For this purpose, we consider a simple and well understood example, namely an elevator in a building with four floors. We have the following type declarations.

```
data Floor  = 0..3;
data Level  = record {set:bool, reset:bool,req:bool};
data Levels = array Floor of Level;
data Goal   = Floor union { nogoal };
data Dir    = up,down,none;
data Door   = open,opening,closed,closing;
data Ppd    = nobody,somebody,vip;
data State = record {levels : Levels,
                     pos    : Floor,
                     goal   : Goal,
                     dir    : Dir,
                     door   : Door,
                     ppd    : Ppd};
data Lift  = array nat of State;
```

The type `Lift` models all possible traces of the elevator and a trace is an array of states. A state of the elevator contains several informations: the current requirements sent from the different floors; the current position, goal and direction; the state of the door and the kind of person within the elevator.

Initially, the elevator has no request. It resides in the first floor, its target is `nogoal` and its direction is `none`. The door is open and nobody is in the elevator.

```
pred init(s:State) =
        all Floor p: ~ s.levels[p].req &
        s.pos   = 1      &
        s.goal  = nogoal &
        s.dir   = none   &
        s.door  = open   &
        s.ppd   = nobody;
```

Now, let us describe how the elevator makes a move. We have to state how the different state components (`pos`, `dir`, ...) change their values during a move. We restrict ourselves to describing only how the position changes; the other components are handled in the same manner.

```
pred next_pos(s,t:State) =
   if ~stop(s)
     then
        case s.dir  of
             up   => t.pos = s.pos + 1;
             down => t.pos = s.pos - 1;
             none => t.pos = s.pos;
        esac
   else t.pos = s.pos;
```

The variables s and t are of type `State` and s and t model the current and next state of the elevator respectively. The predicate `stop` holds if the elevator is stopped (it is not moving). The position changes accordingly to the direction of the elevator if it is moving.

The entire transition relation `next` of the elevator is the conjunction of the next relation of the several components. The valid runs of the elevator are described by the following predicate `run`.

```
pred run (L:Lift) = init(L[0]) & all nat i: 0<i -> next(L[i-1],L[i]);
```

We formalize now the property that the elevator never moves with the door open.

```
pred spec(L:Lift) = all nat i: L[i].dir ~=none -> L[i].door =closed;
```

We can check if the previous specification holds for every run of the elevator.

```
var  Lift L;
run(L) -> spec1(L);
```

The whole formalization of the elevator system takes about 3 pages in LLISA (Appendix C.1) and its WS1S encoding is about 20 pages (Appendix C.2). □

3.7 Related work

In Section 3.1.1, we already described the FIDO approach and used it as starting point and motivation for LISA. We now report on other approaches that also are close to our work.

3.7.1 FMONA

FMONA [BF00a, BF00b] is a high-level interface for MONA. Similarly to LLISA, FMONA enriches the linear fragment (WS1S) of MONA with structured types and high-level programming primitives. Thanks to the higher-order facilities, in FMONA system of finite states, and even parameterized systems of infinite states, can be comfortably specified. For the automatic verification purposes, FMONA allows the instantiations of the parameterized infinite systems with concrete values and translates them into systems of finite states that are compiled into WS1S and decided by MONA.

Another capability of FMONA consists in providing a declarative way to specify validation techniques that are used to verify the specified system. These techniques include *abstraction* [BLO98, CGL94] and iteration (*forward* and *backward search* with *acceleration*[8] [BF00b, PS00]). For instance, if the user provides a description of an infinite system together with an abstraction relation, then FMONA generates a finite system according to the abstraction relation and forwards it to MONA to decide it.

FMONA can be seen as macro-preprocessor for MONA and it has not an own semantics. This permits understanding FMONA specifications only via (as it is also the case for FIDO) the translation into WS1S. Furthermore, the lack of a semantics of FMONA makes it impossible to speak about a correct compilation of FMONA code into any other formalism (such as WS1S).

3.7.2 Guided Tree Automata and WRST

The notion of *guided tree automata* (GTA) is introduced in [BKR97] as a mathematical concept that can replace the ordinary bottom-up tree automata in the decision procedure of WS2S.

A GTA is a bottom-up tree automaton where the set of states (state space) is partitioned in different disjoint sets and where the transition relation respects this partitioning. The splitting of the state space in regions allows for an independent traversal of the subtrees of the input tree. In other words, a GTA is a family of bottom-up tree automata that together with a *guide* act as an ordinary bottom-up tree automata. The guide, which is itself a top-down tree automata, first determines which automaton has to process which subtree and second defines how the results of these automata have to be combined to decide the acceptance or rejection of the input tree. The recent versions of the MONA system make use of the GTA concepts and generate from a formula in WS2S a GTA, where the guide is supplied by the user. The guide for a GTA requires knowledge about the application domain.

GTA is a technical concept introduced with the primary efficiently represent the transition relation of tree automata and thus to alleviate generally the complexity costs in practice. The implementation of the LISA prototype does not make use of this concept; this could be the subject of future work. In the following, we describe some successful applications that use the GTA concept available in MONA.

[8]Acceleration is a technique that allows to group a number of (local) transition steps in a transition system to form a single (accelerated) transition step.

The paper [EMS00] introduced the type system *WRST* that is now available in the MONA system. The logic *WRST* stands for "Weak Second-order logic with Recursive Types". This logic extends the language of WS2S with recursive types and some high-level structural primitives. Types in *WRST* have the grammar

$$\mathit{type}\ E \;=\; V_1^1(C_1^1{:}E_1^1, \ldots, C_{m_1}^1{:}E_{m_1}^1), \ldots, V_n(C_1^n{:}E_1^n, \ldots, C_{m_n}^n{:}E_{m_n}^n)$$

The V_i's are called variants, the C_i's are called components, and the E_i's are declared types.

Formulae in *WRST* are built in the same way as in WS2S. First-order variables range over positions in trees and second-order variables range over sets of positions. Thus, in contrast to LISA, it is not a logic over trees in the direct sense.

Despite from the fact that in LISA we use the terminology of labels instead of variants and of features instead of components, the LISA system is more general, it is WS2S complete whereas the *WRST* type system is not. Furthermore, the LISA system provides a more liberal syntax. However, the biggest improvement in this work compared to LISA is that the translation into WS2S is based on the GTA and the guides of the GTAs are automatically generated and lead often to efficient GTAs.

3.7.3 Parsing with Logical Constraints

The next two related works are about formalizing context-free grammars augmented with side conditions. The basic idea behind these works is that the context-free grammars can be translated into types à la FIDO, *WRST*, or also à la LISA and the side conditions are expressed in a language very close to WS2S. These works have same similarities with the approach of James Rogers [Rog94] who has designed on the top of WS2S a language for expressing parse tree constraints.

Design Constraints for Corba Software systems are in general built in system platforms that are subject to design constraints which should be ensured by the programmers. Design constraints are often captured informally as guideline documents, a fact that makes them difficult to be formally check. Klarlund et al. [KKS96] have showed how a large class of design constraints for Corba can be formalized and automatically verified. The constraints are expressed in a language called *CDL* which is translated into tree automata by means of the FIDO compiler and the MONA decision procedure. The generated automaton is applied to the program parse tree to check if the constraints hold.

YakYak, a Preprocessor for Yacc Conventionally, the syntax of programming languages is described by context-free grammars where the productions are restricted by side constraints that are specified as action code (as is the case in yacc or bison). YakYak [DKS99] is an improvement of yacc in that it extends yacc with a specification language in which the side constraints can be expressed declaratively in a concise and simple way. The specification language is the first-order fragment of WS2S. The input file of YakYak contains the grammar augmented with formulae stating conditions to the parse tree. YakYak forwards the formulae to Mona to turn them into bottom-up tree automata. Afterwards, the automata are translated into C code that are inserted in the productions instead of the formulae. The resulting file contains ordinary yacc code. The generated parser works now as follows: whenever it makes a reduction step the automata make the corresponding transitions and the truth of the side constraints depends on whether the current states of the automata are acceptance states.

3.8 Chapter Summary

Our contribution is to provide a new way for users to estimate the complexity of their specification, namely the size of the resulting compiled tree automaton. Type compilation has an exponential upper bound that in many practical cases is linear. Hence, the more a system can be specified with types, the more accurately one can bound the complexity. Of course, for the part of the system specified in the kernel logic, non-elementary blow-ups (an exponential blowup with each quantifier alternation) are, of course, still possible. There are interesting practical tradeoffs here: as noted in the introduction, all tree automata can be described using types and hence it follows that there are certain problems that can be more naturally (and, in particular, with a non-elementary savings of space) described in the kernel Lisa logic.

Lisa may be the first specification language compiled into the logic WS2S. The language is still primitive, and one can think of extensions that further help in structuring specifications. For example, one could replace the primitive `pred` construct with more powerful means of decomposing specifications into modules that support abstraction and specification reuse.

Chapter 4

Bounded Model Construction

In this chapter we investigate procedures for bounded model construction for monadic logics on finite words as well as on infinite words. For monadic second-order logics on finite words, the problem is, given a formula ϕ and a natural number k, does there exist a word model for ϕ of length k. We give a bounded model construction algorithm for M2L-Str *that runs in a time exponential in k. For* WS1S*, we prove a negative result: bounded model construction is as hard as validity checking, i.e. it is at least non-elementary. From this, negative complexity results for other monadic logics, such as* S1S*, follow. We show furthermore that by allowing quantification over singleton sets in* WS1S *and* S1S*, the bounded model construction becomes elementary.*

4.1 Introduction

Despite the numerous success stories of system verification based on monadic logics, there are limitations on the use of this verification method in terms of the size of the system. Not surprisingly, many large systems cannot be verified due to state-space explosion. This is analogous to state-space explosion in model checking where the state-space is exponential in the number of state variables, except for monadic logics the number of states in the constructed automaton can be non-elementary in the size of the input formula! For LTL model checking, a way of finessing this problem has recently been proposed: *bounded model checking* [BCCZ99]. The idea is that one can finitely represent counter-examples (using the idea of a loop, see Definition 4.3.1), and, by bounding the size of these representations, satisfiability checkers can be used to search for them. This often succeeds in cases where symbolic model checking fails.

Motivated by the bounded model checking approach and the goal of quick generation of counter-examples for falsifiable monadic formulae, we investigate an analogous problem for monadic logics. Namely, given a formula ϕ and a natural number k, determine if ϕ has a word model of length k. Since we are concerned with *constructing* models for formulae, as opposed to *checking* their satisfiability with respect to a given model, we call our problem *bounded model construction* or BMC for short.

The chapter is organized as follows. In Section 4.2 we explore the bounded model construction problem for several monadic second-order logics on finite words. In Section 4.3 we explore the bounded model construction problem for several monadic second-order logics on infinite words. In Section 4.4 we summarize our results.

4.2 BMC for Monadic Logics on Finite Words

We distinguish between the monadic logics on finite words and the monadic logics on infinite words and treat the bounded model construction for these two kinds of logics separately. These differentiation appears to be natural, as these two kind of logics differs on the size of the models. For monadic logics on finite words, models are finite words and therefore the meaning of the term *bounded model* seems to be clear and intuitive. While for monadic logics on infinite words models are infinite words and therefore the meaning of the above term needs clarification.

In this section we present the bounded model construction approach for the logics M2L-Str, WS1S and their first-order fragments FO-Str[<], FO-Str[+], WFO[<], and WFO[+] (see Section 4.2.4). We show that for the logics M2L-Str, FO-Str[<], FO-Str[+], and WFO[+] the bounded model construction can generate counter-examples for non-theorems non-elementary faster than its automata-theoretic counterpart presented in Section 2.4.4. For the other two logics we show a negative result; that is we prove that the bounded model construction problem is as hard as checking validity, which is non-elementary.

The problem we analyze is, how to generate counter-examples of a given size and do this quickly (elementary!) with respect to the size parameter. We express this in the format of a parameterized complexity problem (*cf.*, [AEFM89]). For L either M2L-Str, WS1S, FO-Str[<], FO-Str[+], WFO[<], or WFO[+], we define:

Definition 4.2.1
Bounded Model Construction for L *(*BMC(L)*)*
Instance: *A formula* ϕ *and a natural number* k.
Parameter: k.
Question: *Does* ϕ *have a satisfying word model of length* k *with respect to* L*? (That is, is there a word* w *of length* k *with* $w \models_{\mathsf{L}} \phi$*?)*

We want first to point out that we can of course use the automata-based decision procedures, to solve the bounded model construction for L. Namely, for a given formula ϕ and a bound k, we build the automaton A_ϕ corresponding to the formula ϕ and then check for accepting words of the size k. This method, however, fails to achieve our goal, as the constructed automaton A_ϕ or the intermediate automata needed for its construction could be non-elementary in the size of ϕ. Thus, this method is generally time and space consuming if it ever succeeds. Our intention is now to look for alternatives to these automata-based techniques.

4.2.1 BMC for M2L-Str

We show that for M2L-STR, given a formula ϕ and a natural number k, we can generate a formula in quantified Boolean logic that is satisfiable if and only if ϕ has a word model of length k. The formula generated is polynomial in the size of ϕ and k and can be tested for satisfiability in polynomial space. For generating counter-models of length k, this yields a non-elementary improvement over the automata-based decision procedure for M2L-STR.

We proceed by defining a family of functions $(\lceil . \rceil_k)_{k\in\mathbb{N}}$ that transform MSO-formulae into quantified Boolean formulae such that there is a word model of length k for ϕ iff $\lceil\phi\rceil_k$ is satisfiable. The size of the resulting formula is polynomial in the size of ϕ and k.

To simplify matters, we reduce MSO to its minimal kernel MSO$_0$ using the simple polynomial time translation explained in Section 2.4.4. Recall, MSO$_0$ has the following grammar.

$$\phi ::= \mathsf{Succ}(X,Y) \mid X \subseteq Y \mid \neg\phi \mid \phi \vee \phi \mid \exists X.\, \phi, \qquad X, Y \in \mathcal{V}_2\ . \tag{4.1}$$

Translation to Quantified Boolean Logic

Let $k \in \mathbb{N}$ be fixed. We now describe how to calculate the QBL formula $\lceil\phi\rceil_k$ for a MSO$_0$-formula ϕ. The idea is simple: a set $M \subseteq [k]$ can be represented by k Boolean variables $x_0, \ldots, x_{k-1}$ such that $x_i = \mathsf{true}$ iff $i \in M$. Building on this, we encode relations between finite sets and formulae over these relations.

Let $\mathcal{V}_0 = \{x^i_j \mid i, j \in \mathbb{N}\}$ be a set of Boolean variables and let $\mathsf{singleton}$ be the Boolean formula

$$\mathsf{singleton}(x_0, \ldots, x_{k-1}) \overset{def}{=} \bigvee_{0\leq i\leq k-1} (x_i \wedge \bigwedge_{\substack{0\leq j\leq k-1 \\ j\neq i}} \neg x_j)\ .$$

The mapping $\lceil . \rceil_k$ is inductively defined as follows:

Definition 4.2.2 (Translation)

$$\begin{array}{lcl}
\lceil X_m \subseteq X_n \rceil_k & = & \bigwedge_{0 \leq i \leq k-1}(x_i^m \rightarrow x_i^n) \\
\lceil \mathsf{Succ}(X_m, X_n) \rceil_k & = & \mathsf{singleton}(x_0^m, \ldots, x_{k-1}^m) \wedge \\
 & & \mathsf{singleton}(x_0^n, \ldots, x_{k-1}^n) \wedge \\
 & & \bigvee_{0 \leq i < k-1}(x_i^m \rightarrow x_{i+1}^n) \\
\lceil \phi_1 \vee \phi_2 \rceil_k & = & \lceil \phi_1 \rceil_k \vee \lceil \phi_2 \rceil_k \\
\lceil \neg \phi \rceil_k & = & \neg \lceil \phi \rceil_k \\
\lceil \exists X_m .\, \phi \rceil_k & = & \exists x_0^m, \ldots, x_{k-1}^m .\, \lceil \phi \rceil_k
\end{array}$$

Definition 4.2.3 *For a substitution $\sigma{:}\mathcal{V}_2 \rightarrow \mathsf{Pow}(\mathbb{N})$, we define the Boolean substitution $\widehat{\sigma}{:}\mathcal{V}_0 \rightarrow \mathbb{B}$, by $\widehat{\sigma}(x_i^m) = 1$ iff $i \in \sigma(X_m)$.*

Lemma 4.2.4 *Let σ be a substitution and $k \in \mathbb{N}$. Then $\sigma^k \models_{\scriptscriptstyle M2L} \phi$ iff $\widehat{\sigma} \models_{\scriptscriptstyle QBL} \lceil \phi \rceil_k$.*

Proof By induction on the construction of ϕ.

We first establish the claim for atomic formulae. To begin with,

$$\sigma^k \models_{\text{M2L}} X_m \subseteq X_n \text{ iff for all } i, 0 \leq i \leq k-1, i \in \sigma(X_m) \text{ implies that } i \in \sigma(X_n),$$

which is equivalent to

$$\widehat{\sigma} \models_{\text{QBL}} \bigwedge_{0 \leq i \leq k-1} x_i^m \rightarrow x_i^n .$$

Similarly, if $\sigma^k \models_{\text{M2L}} \mathsf{Succ}(X_m, X_n)$, then $\sigma(X_m)$ and $\sigma(X_n)$ are singletons. Moreover, $\sigma(X_m)$ contains a natural number p, with $0 \leq p < k-1$, whose successor $p+1$ is in $\sigma(X_n)$. Hence there is some i, where $0 \leq i < k-1$, such that $\widehat{\sigma} \models_{\text{QBL}} x_i^m \rightarrow x_{i+1}^n$, so $\widehat{\sigma} \models_{\text{QBL}} \bigvee_{0 \leq i < k-1} x_i^m \rightarrow x_{i+1}^n$; the converse is argued similarly.

In the inductive step we consider only the case where ϕ is of the form $\exists X_m .\, \psi$ as the remaining cases are straightforward. By Definition 2.4.1, $\sigma^k \models_{\text{M2L}} \exists X_m .\, \psi$ iff there is some set $M \subseteq [k]$ such that $(\sigma[M/X_m])^k \models_{\text{M2L}} \psi$. From the induction hypothesis,

$$(\sigma[M/X_m])^k \models_{\text{M2L}} \psi \text{ iff } \widehat{\delta} \models_{\text{QBL}} \lceil \psi \rceil_k, \text{ where } \delta = \sigma[M/X_m].$$

Note that $\widehat{\delta} = \widehat{\sigma}[b_0/x_0^m, \ldots, b_{k-1}/x_{k-1}^m]$, where $b_i = 1$ iff $i \in M$, for $0 \leq i \leq k-1$. Further, $\widehat{\delta} \models_{\text{QBL}} \lceil \psi \rceil_k$ iff $\widehat{\sigma} \models_{\text{QBL}} \exists x_0^m, \ldots, x_{k-1}^m .\, \lceil \psi \rceil_k$. Thus,

$$\sigma^k \models_{\text{M2L}} \exists X_m .\, \psi \text{ iff } \widehat{\sigma} \models_{\text{QBL}} \exists x_0^m, \ldots, x_{k-1}^m .\, \lceil \psi \rceil_k .$$

■

Observe that given a Boolean substitution τ, it is trivial to define an MSO substitution σ where $\widehat{\sigma} = \tau$, namely by stipulating that $\sigma(X_i) = \{j \mid \tau(x^i_j) = 1\}$. Hence, from the above Lemma we can conclude:

Theorem 4.2.5 (Correctness) *Let ϕ be a* MSO *formula. For $k \in \mathbb{N}$, there exists a* MSO *substitution σ where $\sigma^k \models_{M2L} \phi$ iff there exists a Boolean substitution τ where $\tau \models_{QBL} \lceil\phi\rceil_k$. Moreover, ϕ is valid in* M2L-STR *iff for all $k \geq 0$, the* QBL *formula $\lceil\phi\rceil_k$ is valid.*

We can avoid the intermediate translation of MSO into MSO$_0$ in the above algorithm and give rather a more direct translation that also handles first-order quantification. This translation that we denote with $[.]_k$, is defined for first-order quantification by the following rule

$$[\exists p.\, \phi]_k \;=\; \bigvee_{0 \leq i \leq k-1} [\, \phi[p/i]\,]_k,$$

and it has the same rules as $\lceil.\rceil_k$ for the other cases of formulae.

We can prove then that for any MSO formula ϕ, $[\phi]_k$ and $\lceil\phi\rceil_k$ are semantically equivalent. The formula $[\phi]_k$ could be in general exponentially larger than $\lceil\phi\rceil_k$. However, in the translation $\lceil.\rceil_k$ first-order quantifications involves Boolean quantifications, because they are first shifted to second-order quantifications, whereas this is not the case in $[.]_k$.

Recall that the size of a formula (in any of the logics we consider) is defined as the number of symbols occurring in its string representation. Exploiting the fact that satisfiability for QBL is *PSPACE-complete*, we prove:

Theorem 4.2.6 (Complexity) BMC(M2L-STR) *is* PSPACE-complete.

Proof Let ϕ and k be a problem instance. The size of $\lceil\phi\rceil_k$ is $O(k^2|\phi|)$. It follows that BMC(M2L-STR) can be reduced in polynomial time to satisfiability in QBL, which establishes membership in *PSPACE*.

To prove *PSPACE-hardness*, we show that satisfiability for QBL can be reduced in log-space to BMC(M2L-STR). Let E be a fresh second-order variable and empty be the M2L-STR proposition defined by

$$\mathsf{empty}(X) \overset{def}{=} \forall Y.\, X \subseteq Y.$$

We encode each Boolean variable x with a M2L-STR variable X. For a QBL formula ϕ, let $\widetilde{\phi}$ be the M2L-STR formula obtained from ϕ as follows: replace occurrences of Boolean variables x by $X \subseteq E$, and replace the Boolean quantifiers as well as the propositional connectives by the corresponding quantifiers and connectives of M2L-STR. Now, the encoding of ϕ in M2L-STR is the formula

$$\exists E.\, \mathsf{empty}(E) \wedge \widetilde{\phi}.$$

For example, the QBL formula $\forall x\, \exists y.\, x \vee y$ is encoded as

$$\exists E.\, \mathsf{empty}(E) \wedge \forall X.\, \exists Y.\, X \subseteq E \vee Y \subseteq E.$$

Under this encoding it is only relevant whether or not a second-order variable is interpreted by the empty set. We immediately conclude that a QBL formula is satisfiable iff its encoding has a word model of length 1. ■

Example 4.2.1 To illustrate how BMC works for M2L-STR, let us consider two simple examples. The formula $\forall x.\, x \in X$ has exactly the intervals $[n]$, for $n \in \mathbb{N}$ as models in M2L-STR. The bounded model constructor for M2L-STR applied to this formula and $k \in \mathbb{N}$ produces the Boolean formula $\bigwedge_{0 \leq i \leq k-1} x_i$ which is satisfiable for every k. Note that the empty conjunction is by convention true.

The formula $\exists X.\, \forall p.\, p \in X \to p+1 \in X$ is valid in M2L-STR and the witness for X that makes the formula true is uniquely the empty set. The application of BMC to this formula yields the following quantified Boolean formula

$$\exists x_0, \ldots, x_{k-1}.\, (\bigwedge_{0 \leq i < k-2} x_i \to x_{i+1}) \wedge (x_{k-1} \to \mathsf{false})$$

The subformula false in the last implication of the above formula is due to the fact that the successor relation in M2L-STR is partial. That is, there is no number in the interval $[k-1]$ that builds the successor of $k-1$. The above formula can be simplified into $\exists x_0, \ldots, x_{k-1}.\, \bigwedge_{0 \leq i \leq k-1} \neg x_i$ which is valid for every k by assigning the value false to the variables x_i. □

4.2.2 BMC for WS1S

We now investigate bounded model construction for the monadic logic WS1S and establish a negative result by showing that BMC is as hard as checking validity, which is non-elementary.

The previously given translation cannot be employed for WS1S. If ϕ is the formula $\exists X.\, \forall Y.\, Y \subseteq X$, the translation yields the quantified Boolean formula

$$\exists x_0, \ldots, x_{k-1}.\, \forall y_0, \ldots, y_{k-1}.\, \bigwedge_{0 \leq i \leq k-1} y_i \to x_i,$$

which is valid for every k, whereas ϕ is unsatisfiable in WS1S. We now prove that there is no translation that will yield an elementary bounded model construction procedure.

Theorem 4.2.7 BMC(WS1S) *is non-elementary.*

Proof For a closed formula ϕ, $\sigma \models_{\mathrm{WS1S}} \phi$ iff $\sigma' \models_{\mathrm{WS1S}} \phi$ for all substitutions σ and σ', *i.e.* the satisfiability of a closed formula does not depend on the substitution. Hence, every closed WS1S formula is either valid or unsatisfiable. Equivalently, for ϕ a closed formula, we have either $\mathcal{L}_{\mathrm{WS1S}}(\phi) = ()^*$ or $\mathcal{L}_{\mathrm{WS1S}}(\phi) = \emptyset$. Consequently, if a closed formula ϕ has a word model, then $\mathcal{L}_{\mathrm{WS1S}}(\phi) = ()^*$ and therefore ϕ is valid. In other words, computing a word model of any length for ϕ is equivalent to checking ϕ's validity. ■

4.2.3 Comparing M2L-Str and WS1S

We continue the discussion that we began in Section 2.4.4 using the insights that we gained by investigating the BMC problem for both logics M2L-Str and WS1S.

The result of Theorem 4.2.7 is somewhat surprising since WS1S, as we already stated in 2.4.4, has the same expressiveness and complexity as M2L-Str and their decision procedures differ only slightly. The reader may wonder what causes these differences. We can gain some insight by comparing semantics. From the semantics of M2L-Str, $\phi(X)$ has a word model of length k iff $\exists X.\, \phi(X)$ has a word model of length k. This semantic property was employed in the proof of Lemma 4.2.4, where in order to use the induction hypothesis we require that the witness set M is a subset of $[k]$. Unfortunately, this property fails for WS1S. As can be seen in Figure 2.5, existential quantification can change the size of the minimal word model in WS1S. An example is the family of formulae (written here with sugared syntax) $\phi_n(X) \overset{def}{=} X(n)$, for $n \in \mathbb{N}$. The minimal length word model for $\phi_n(X)$ is n, whereas it is 0 for $\exists X.\, \phi_n(X)$. In general, to determine if a formula has a small, e.g., length 0, word model, we must consider word models for their subformulae that are non-elementary larger in the worst case.

There has been a recent investigation of the differences of these logics by Klarlund who concluded that WS1S is preferable to M2L-Str due to its simpler semantics and its wider applicability to arithmetic problems [Kla99]. Our results suggests that the issue is not so clear cut and depends on whether error detection through counter-example generation versus full verification is desired, that is, whether one is interested in finding a single model for a formula or computing a description of all models.

In [KM01] a linear embedding of M2L-Str in WS1S is given. Reciprocally, it is not known whether an efficient embedding of WS1S in M2L-Str exists. Moreover, we know that there is an efficient compilation of Presburger arithmetic in WS1S, whereas a similar efficient compilation in M2L-Str is to date unknown. This observation may indicate that translations of WS1S into M2L-Str could be not efficient; in other words WS1S could be non-elementary more concise than M2L-Str. In the following, we show that there is no elementary translation ξ from WS1S into M2L-Str such that each formula in WS1S has a model of length k iff its image $\xi(\phi)$ in M2L-Str has a model of length elementary in k.

Lemma 4.2.8 *There are no elementary mapping* $\xi : \mathrm{MSO} \to \mathrm{MSO}$ *and* $f : \mathbb{N} \to \mathbb{N}$ *such that for each formula* ϕ *and for every natural number* k*:* ϕ *has in* WS1S *a model of length* k *iff* $\xi(\phi)$ *has in* M2L-Str *a model of length* $f(k)$*.*

Proof By contradiction. Assume that such a mapping ξ and a mapping f exist and let ϕ be a closed MSO formula. Then by the semantics of WS1S, ϕ is valid in WS1S iff it has a model of length 1. Now, by assumption ϕ is valid in WS1S iff $\xi(\phi)$ has a model of length $f(1)$. By Lemma 4.2.6, checking if $\xi(\phi)$ has a model of length $f(1)$ is in *PSPACE* with respect to the size of $\xi(\phi)$ and thus the validity check of ϕ in WS1S is elementary in the size of ϕ, which contradicts the fact that validity check in WS1S is non-elementary in the size of the input formula. ■

Based on the above Lemma and the fact that WS1S can linearly be embedded in S1S (*cf.* Section 4.3.1) we conclude that there is no efficient encoding of S1S into M2L-Str.

4.2.4 BMC for the First-Order Fragment of MSO

In this section we incrementally decrease the expressive power of M2L-Str and WS1S and investigate how in relationship to this the results concerning BMC varies. We consider several first-order logics on finite words. The first-order language (FOL) is specified by the following grammar:

$$\begin{aligned} t &::= \mathsf{0} \mid p \mid \mathsf{s}(t,t), && p \in \mathcal{V}_1 \\ \phi &::= t = t \mid t < t \mid X(t) \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\, \phi, && p \in \mathcal{V}_1 \text{ and } X \in \mathcal{V}_2 \end{aligned}$$

Observe that FOL can be seen as a sublanguage of MSO, as it can be obtained from MSO by (i) removing second-order quantification and (ii) adding the new primitive relation $<$. The relation $<$ intuitively denotes the less relation over the natural numbers and it is definable in MSO, as we have seen in Section 2.4.4. Moreover, it is known [Str94, Tho90] that $<$ is not definable in MSO, if no use of second-order quantification is made. Hence $<$ is explicitly included as a part of the syntax in FOL.

We call FO-STR[$<$] and WFO[$<$] the logics obtained by restricting M2L-STR and WS1S respectively to FOL and we call FO-STR[$+$] and WFO[$+$] the logics obtained by restricting M2L-STR and WS1S respectively to FOL without $<$.

Example 4.2.2 Let us consider some simple examples to emphasize the differences between these logics.

(1) The formula $\forall p.\, p \in X$ is satisfiable in FO-STR[$+$] and unsatisfiable in WFO[$+$].

(2) The formulae $\forall x.\, \exists y.\, y = x + 1$ and $\forall x.\, \exists y.\, x < y$ are both valid in WFO[$<$], but both unsatisfiable in FO-STR[$<$].

(3) The formula $0 \in X \wedge \forall p.\, p \in X \to p{+}1 \in X$ is unsatisfiable in both FO-STR[$<$] and WFO[$<$]

□

As we mentioned before, disallowing second-order quantification in MSO affects the expressiveness of the logics M2L-STR, and WS1S. Namely, it can be shown that the expressiveness of FO-STR[$<$] and WFO[$<$] coincides with the star-free regular languages (*cf.*, Section 2.2.1). Here also again, similarly to M2L-STR and WS1S, we can establish a one-to-one connection between FO-STR[$<$] and star-free regular languages, whereas the connection between WFO[$<$] and this class of languages is up to 0-padding. The expressive power of the logics FO-STR[$+$] and WFO[$+$] is even more restrictive. It is shown [Str94, Tho97] that these logics are as expressive as locally threshold testable sets which forms a proper subclass of the star-free regular languages.

We now show that although the substantial reduction on the expressive power, regarding BMC, we have for the fragments FO-STR[$<$], FO-STR[$+$] and WFO[$<$] similar results as for their super logics M2L-STR and WS1S.

Theorem 4.2.9 BMC(FO-STR[$<$]) *and* BMC(FO-STR[$+$]) *are* PSPACE-complete.

Proof The membership in *PSPACE* follows immediately form Theorem 4.2.6, as FO-STR[$<$] and FO-STR[$+$] are sublogics of M2L-STR.

For *PSPACE-hardness*, we show that the satisfiability of QBL can be reduced in log-space in BMC(FO-STR[$+$]). A quantified Boolean formula is ϕ is translated to the MSO formula $(\exists x.\, x = 0) \wedge \xi(\phi)$, where the mapping ξ: QBF $\to$ MSO, is defined by

$$\begin{array}{lcl}\xi(x) & = & x = 0 \\ \xi(\neg\phi) & = & \neg\xi(\phi) \\ \xi(\phi_1 \vee \phi_2) & = & \xi(\phi_1) \vee \xi(\phi_2) \\ \xi(\exists x.\, \phi) & = & \exists x.\, \xi(\phi)\end{array}$$

We can prove inductively that a quantified Boolean formula ϕ is satisfiable iff its encoding has a word model of length 1. ■

Meyer showed in [Mey75] that WFO[<] is also non-elementary; based on his result, we can easily adapt the argument given in the proof of Theorem 4.2.7 to show that bounded model construction for WFO[<] is non-elementary.

Theorem 4.2.10 BMC(WFO[<]) *is non-elementary.*

The result of BMC for WFO[+] is provided in the next section.

4.3 BMC for Monadic Logics on Infinite Words

In this section we study the bounded model construction for monadic logics over infinite words. We consider the logic S1S as well as its restriction to FOL, FO[<] and its restriction to FOL without <. The expressiveness of FO[<] coincides with the star-free ω-languages [Kam68a, LPZ85, Tho89], which is a proper subclass of ω-regular languages. The expressiveness of FO[+] coincides with locally threshold testable sets of infinite words, which forms a proper subclass of the star-free regular languages.

Because models in these logics are *infinite* words (and thus *unbounded*), we cannot adopt the definition of bounded model construction from the previous section. In order to appropriately define the bounded construction here and establish results, we provide first some background on ω-regular languages and show how certain infinite words can be represented using finite words.

Definition 4.3.1 (Lasso-Words) *Let u and v be finite words in Σ^*. We say that the word uv^ω is* a $(|u|, |uv|)$-lasso, *with* prefix u, loop v, *and length $|uv|$.*

We recall that ω-regular languages are the languages recognizable by Büchi automata. From the definition of Büchi acceptance condition, follows that every nonempty ω-regular language contains a lasso word.

Proposition 4.3.2 *A Büchi automaton $\mathcal{A}$ accepts an ω-word π iff it also accepts the (l, k)-lasso word $\pi_0 \dots \pi_{l-1}(\pi_l \dots \pi_{k-1})^\omega$, for some $l, k \in \mathbb{N}$ with $l < k$.*

Now, let L be either S1S, FO[<] or FO[+], ϕ be a formula in L, and $\mathcal{L}_{\mathsf{L}}(\phi)$ is the ω-regular language denoted by ϕ. By Proposition 4.3.2, the satisfiability of ϕ in L can be reduced to finding a lasso word π in $\mathcal{L}_{\mathsf{L}}(\phi)$. The bounded model construction problem for monadic second-order logics over infinite words is defined as follows:

Definition 4.3.3
Bounded Model Construction for L *(*BMC(L)*)*
INSTANCE: *A formula* ϕ *and a natural number* k*.*
PARAMETER: k*.*
QUESTION: *Does* ϕ *have a satisfying lasso of length* k *in* L*?*

4.3.1 BMC for S1S

We prove that no elementary BMC procedure for S1S exists. To do this, we give first an embedding of WS1S in S1S. Let Finite and finite be the following two propositions:

$$\mathsf{finite}(X) \overset{def}{=} \exists m.\, \forall p.\, X(p) \to p \leq m \text{ and } \mathsf{Finite}(\phi) \overset{def}{=} \bigwedge_{X \in freevars(\phi)} \mathsf{finite}(X)\,.$$

Definition 4.3.4 (Embedding of WS1S in S1S)
We define an embedding function $[\cdot]$ *from* WS1S *into* S1S *by* $[\phi] = \mathsf{Finite}(\phi) \wedge \lceil\phi\rceil$*, where* $\lceil\phi\rceil$ *is:*

$$\lceil \mathrm{s}(t,t') \rceil = \mathrm{s}(t,t') \qquad \lceil X(t) \rceil = X(t) \qquad \lceil \exists p.\, \phi \rceil = \exists p.\, \lceil \phi \rceil$$
$$\lceil \neg\phi \rceil = \neg\lceil \phi \rceil \qquad \lceil \phi_1 \vee \phi_2 \rceil = \lceil \phi_1 \rceil \vee \lceil \phi_2 \rceil \qquad \lceil \exists X.\, \phi \rceil = \exists X.\, \mathsf{finite}(X) \wedge \lceil \phi \rceil$$

Lemma 4.3.5 *Let* ϕ *be a formula and* σ *an* S1S*-substitution. Then it holds:* $\sigma \models_{\mathrm{S1S}} [\phi]$ *iff* σ *is an* WS1S*-substitution and* $\sigma \models_{\mathrm{WS1S}} \phi$*.*

Proof By structural induction over ϕ.

We first establish the claim for atomic formulae. To begin with, by definition of $\models_{\mathrm{S1S}}$ and [.], $\sigma \models_{\mathrm{S1S}} [X(t)]$ is equivalent to $\sigma(X)$ is finite and $\sigma \models_{\mathrm{WS1S}} X(t)$, which is by definition of $\models_{\mathrm{WS1S}}$ equivalent to $\sigma \models_{\mathrm{WS1S}} X(t)$.

In the inductive step we consider only the case where ϕ is of the form $\neg\phi'$ or $\exists X.\, \phi'$ the remaining cases are straightforward. We consider the case where ϕ is of the form $\neg\phi'$. First, notice that $[\neg\phi]$ is equivalent to $\mathsf{Finite}(\phi') \wedge \neg\lceil\phi'\rceil$. Now, $\sigma \models_{\mathrm{S1S}} [\neg\phi']$ is equivalent to $\sigma \models_{\mathrm{S1S}} \mathsf{Finite}(\phi)$ and $\sigma \not\models_{\mathrm{S1S}} \phi'$. The goal follows now by the induction hypothesis. We consider now the case where ϕ is of the form $\neg\exists X.\, \phi'$. Similarly, we notice that $[\exists X.\, \phi]$ is equivalent to $\exists X.\, [\phi']$. It follows $\sigma \models_{\mathrm{S1S}} [\exists X.\, \phi]$ is equivalent $\sigma[M/X] \models_{\mathrm{S1S}} [\phi']$ for some $M \subseteq \mathbb{N}$. By induction hypothesis, $\sigma[M/X]$ is an WS1S-substitution and $\sigma[M/X] \models_{\mathrm{WS1S}} \phi'$, which is equivalent to $\sigma \models_{\mathrm{WS1S}} \phi$. ■

Theorem 4.3.6 BMC(S1S) *is non-elementary.*

Proof By Lemma 4.3.5, a formula ϕ has a word model of length k in WS1S iff $[\phi]$ has a satisfying lasso of length k in S1S. The function that assigns to each BMC(WS1S)-instance (ϕ, k) the BMC(S1S)-instance $([\phi], k)$ reduces, in polynomial time, BMC(WS1S) to BMC(S1S). Using Theorem 4.2.7, the claim follows. ■

4.3.2 BMC for FO[<]

We prove also a negative result for FO[<] concerning the BMC problem. We adopt the same argumentation used for S1S, except a minor change in the mapping $\lceil . \rceil$ of Definition 4.3.4 is needed. We extend $\lceil . \rceil$ to handle the predicate $<$ by the rule $\lceil t_1 < t_2 \rceil = t_1 < t_2$ and we obtain an embedding of WFO[<] in FO[<]. From the fact that BMC(WFO[<]) is non-elementary follows that BMC(FO[<]) is non-elementary too.

Theorem 4.3.7 BMC(FO[<]) *is non-elementary.*

The next section is devoted to prove that BMC for WFO[+] as well as for FO[+] are *PSPACE-complete*. The proof is technical and divided in intermediate lemmas whose proofs are for readability reasons given in Appendix B.

4.3.3 BMC for FO[+]

In Section 4.2.3, we argued that the different treatment of second-order quantification in M2L-STR and WS1S is the direct source for the different behavior of these logics regarding BMC. The same argument explains why FO-STR[<] and WFO[<] also behave differently. Then the use of the relation $<$ in these first-order logics implicitly induces the use of second-order quantification. Now, we show that if we disallow the use of the relation $<$ in these logics, the BMC problem for the obtained logics WFO[+] and FO[+] becomes *PSPACE-complete*.

We proceed as follows: we adapt the lasso notion defined for infinite words to substitutions and define then the *bounded semantics* for S1S. In this semantics, we use a finite prefix of a substitution σ to determine the satisfiability of a formula ϕ over σ. We show that the bounded semantics and the original semantics for

S1S coincide for lasso substitution. We define a family of functions $(\lceil . \rceil_k)_{k \in \mathbb{N}}$ that transforms formulae in FO[+] into quantified Boolean formulae such that there is a lasso word of length k that satisfies ϕ if and only if $\lceil \phi \rceil_k$ is satisfiable. Finally, we put all these ingredients together to prove our desired goal.

In Section 2.4.5 we have established the connection between S1S and the class of ω-regular languages and we have seen that S1S substitutions can be seen as infinite words. So, an S1S substitution is said to be *lasso*, if its word encoding is lasso.

Definition 4.3.8 *Let l and k be two natural numbers with $l < k$. We define the function $\langle . \rangle_k^l : \mathbb{N} \to \mathbb{N}$, by*

$$\langle n \rangle_k^l = \begin{cases} n, & \textit{if } n < l \\ l + ((n-l) \ \mathsf{mod}\ (k-l)), & \textit{otherwise.} \end{cases}$$

Definition 4.3.9 (Lasso Set) *A set $M \subseteq \mathbb{N}$ is called (l,k)-*lasso*, if $m \in M$ iff $\langle m \rangle_k^l \in M$.*

Definition 4.3.10 (Lasso Substitution) *A substitution σ is called (l,k)-lasso, if $\sigma(X)$ is a (l,k)-lasso set, for all second-order variable X for which σ is defined.*

Example 4.3.1 Consider the formula $X(0) \wedge \forall p.\, X(p) \leftrightarrow Y(p+1)$ with the free second-order variables X and Y. A substitution satisfying this formula is σ defined by $\sigma(X) = \{2n \mid n \in \mathbb{N}\}$ and $\sigma(Y) = \{0\} \cup \{2n+1 \mid n \in \mathbb{N}\}$. The sets $\sigma(X)$ and $\sigma(Y)$ are $(1,3)$-lasso sets and thus the substitution σ is $(1,3)$-lasso and it can be visualized by the $(1,3)$-lasso word: $\begin{array}{|c|}\hline 1 \\ \hline 1 \\ \hline \end{array} \left(\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & 0 \\ \hline \end{array} \right)^{\omega}$. □

We come now to define the bounded semantics of S1S. This semantics is defined inductively relative to a substitution σ and two natural numbers l and k with $l < k$. Roughly speaking, the rules defining the semantics mimic the evaluation of a formula over the substitution σ, which is assumed to be a (l,k)-lasso.

Definition 4.3.11 (Bounded Semantics of S1S)

$$\begin{array}{llll} \sigma \models_k^l & X(t), & \textit{if} & \langle \sigma(t) \rangle_k^l \in \sigma(X) \\ \sigma \models_k^l & \neg\phi, & \textit{if} & \textit{not } \sigma \models_k^l \phi \\ \sigma \models_k^l & \phi_1 \vee \phi_2, & \textit{if} & \sigma \models_k^l \phi_1 \textit{ or } \sigma \models_k^l \phi_2 \\ \sigma \models_k^l & \exists p.\, \phi, & \textit{if} & \textit{for some } n \in \mathbb{N},\ \sigma[n/p] \models_k^l \phi \\ \sigma \models_k^l & \exists X.\, \phi, & \textit{if} & \textit{there are some } l', k' \in \mathbb{N} \textit{ with } l' < k' \textit{ and} \\ & & & \sigma[M/X] \models_{k'}^{l'} \phi, \textit{ for some subset } M \subseteq \mathbb{N}. \end{array}$$

Lemma 4.3.12 *If σ is a (l,k)-lasso substitution, then $\sigma \models \phi$ iff $\sigma \models^l_k \phi$.*

Remark 4.3.13 We have already established a bounded model construction for S1S is non-elementary. Here, we want to give more informal explanation for this fact. The last rule in Definition 4.3.11 of the bounded semantics of S1S can be read in the following way: checking for the existence of a lasso word of length k that satisfies the second-order quantified formula $\exists X.\, \phi$ could result in a more difficult task, namely to check if the formula ϕ has a lasso word of length possible non-elementary in k. There are examples in which the minimal length of the lasso words satisfying ϕ is in fact non-elementary in k. This means that in certain cases to answer the question if there is a lasso word of length k that satisfies a formula ϕ, we are enforced to answer the same question, however, for a subformula of ϕ and a model length k' that is non-elementary bigger than k.

Lemma 4.3.14 *Let σ be a substitution, ϕ be a* FO[+] *formula, p a first-order variable, and t be a* FO[+] *term containing p. It holds:*

(i) $\sigma[n/p](t) \equiv \sigma[\langle n\rangle^l_k/p](t)\,(\mathsf{mod}\ (k-l))$

(ii) $\sigma[n/p] \models^l_k \phi$ *iff* $\sigma[\langle n\rangle^l_k/p] \models^l_k \phi$*, for all* $n \in \mathbb{N}$.

Translating FO[+] into M2L-Str Below, we describe a family of functions $(\lceil.\rceil_k)_{k\in\mathbb{N}}$ that translate FO[+] formulae into M2L-Str formulae, such that ϕ has a lasso word model of length k iff $\lceil\phi\rceil_k$ has a word model of length k. On the top of this translation we give a linear reduction of the BMC problem for FO[+] into BMC problem for M2L-Str

To define the translation functions mentioned before, we make use of the following two M2L-Str predicate schemata.

$$\begin{aligned}
\mathsf{modulo}(p,q,\Delta) &\overset{def}{=} q < \Delta \wedge \\
&\quad\ \forall X.\,(p \in X \wedge (\forall x.\, x \in X \wedge \Delta \le x \to (x-\Delta) \in X) \to q \in X \\
\mathsf{lasso}(p,q,l,k) &\overset{def}{=} \mathsf{if}\ p < k\ \mathsf{then}\ p = q\ \mathsf{else}\ \mathsf{modulo}(p, q-l, k-l)
\end{aligned}$$

where the parameters p and q are first-order variables and Δ, l, and k are place holder for natural numbers. Let σ be the substitution $[n/p, m/q]$ and Δ, l, and k natural numbers with $l < k$. It holds:

$$\begin{aligned}
&\sigma \models_{\text{M2L}} \mathsf{modulo}(p,q,\Delta) \text{ iff } m = n\,(\mathsf{mod}\,\Delta), \text{ and} \\
&\sigma \models_{\text{M2L}} \mathsf{lasso}(l,k,p,q) \text{ iff } m = \langle n\rangle^l_k\,.
\end{aligned}$$

Definition 4.3.15 *Let k be a natural number. We define $\lceil\phi\rceil_k = \bigvee_{0\le l\le k}\lceil\phi\rceil_k^l$, where the mapping $\lceil.\rceil_k^l$ is defined inductively as follows:*

$$\begin{aligned}
\lceil X(t)\rceil_k^l &= \exists q.\,\mathsf{lasso}(t,q,l,k)\wedge q\in X\\
\lceil\phi_1\vee\phi_2\rceil_k^l &= \lceil\phi_1\rceil_k^l\vee\lceil\phi_2\rceil_k^l\\
\lceil\neg\phi\rceil_k^l &= \neg\lceil\phi\rceil_k^l\\
\lceil\exists p.\phi\rceil_k^l &= \exists p.\,\lceil\phi\rceil_k^l
\end{aligned}$$

We state now a property of the help function $\lceil.\rceil_k^l$ from the above definition.

Lemma 4.3.16 *If σ is a (l,k)-lasso substitution and ϕ a* FO[+] *formula with no free first-order variables, then $\sigma\models_k^l\phi$ iff $\sigma^k\models_{M2L}\lceil\phi\rceil_k^l$.*

The translation $\lceil.\rceil_k$ is satisfiability preserving.

Lemma 4.3.17 *Let ϕ be a* FO[+] *sentence. Then ϕ is satisfiable iff there is some $k\in\mathbb{N}$ such that $\lceil\phi\rceil_k$ is satisfiable.*

Proof By Proposition 4.3.2, the formula ϕ is satisfiable iff there is a lasso word in $L(\phi)$. Because the one-to-one correspondence between words in $L(\phi)$ and substitutions of ϕ, the formula ϕ is satisfiable iff there is a (l,k)-lasso substitution σ with $\sigma\models\phi$. Furthermore, by Theorem 4.3.12, $\sigma\models\phi$ iff $\sigma\models_k^l\phi$. The claim follows now by Lemma 4.3.16. ∎

Lemma 4.3.18 *The bounded model construction for* FO[+] *can be linearly reduced to the bounded model construction for* M2L-Str.

Proof By Lemma 4.3.17, the function that assigns to each BMC(FO[+])-instance (ϕ,k) the BMC(M2L-Str)-instance $(\lceil\phi\rceil_k,k)$ reduces, in linear time, BMC(FO[+]) to BMC(WS1S). Using Theorem 4.2.6, the claim follows. ∎

By the above lemma, the fact that QBL can be encoded in both WFO[+] and FO[+], and the fact that WFO[+] can be embedded in FO[+], we conclude

Theorem 4.3.19 BMC(FO[+]) *and* BMC(WFO[+]) *are* PSPACE-complete.

4.4 Chapter Summary

We have explored the bounded model construction problem for a series of monadic logics on finite words as well as infinite words and we have obtained theoretical and practical contributions. The theoretical contributions are (i) the complexity results for each of the considered logics and (ii) we have used the insights we have gained from these results to shed some light on the differences between the logics M2L-Str and WS1S and to establish a new result. The practical contributions regard the logic M2L-Str.We have obtained a procedure for generating counter-examples that is non-elementary faster than the standard automata-based decision procedures. In Chapter 6 we present an implementation of this procedure and show how we succeed to verify the correctness of a large class of problems whose treatment using the automata-based procedures, like Mona, is unsuccessful.

Chapter 5

LTL Model Checking in M2L-Str

Based on the idea of the finite representation of lasso-words introduced in the previous chapter we show in this chapter how the logic M2L-Str *provides a formalism to reason about finite-state systems with infinite behavior and focus, on the embedding of* LTL *model checking in* M2L-Str. *We prove that the bounded model constructor for* M2L-Str *can be used as a bounded model checker for* LTL *and that it generates the same (up to variable renaming) Boolean formula as the procedure provided by Biere et al. in [BCCZ99].*

5.1 Background and Motivation

Temporal logic model checking of finite-state systems is the task of verifying if a finite-state system obeys a specification of its expected behavior. Finite-state systems are finite-state machines, finite concurrent systems, communication protocols and digital circuits, just to mention a few. System specifications are expressed in formalisms such as the linear temporal logic LTL and the branching temporal logic CTL allowing the reasoning about time events.

A model checking procedure is an algorithm that checks a property of a finite-state system by associating a transition graph to the system and exploring the state-space of this graph afterwards. Model checking algorithms differ mainly in the state representation of the transition graphs. The first model checkers [HK91, BCDM86, CES86] in the early 1980's, used in their implementations an explicit state representation of the transition graphs and the verification process is then based on graph-traversing techniques. The use of these model checkers allowed the automatically discovery of non-trivial errors in circuits and protocols of small size. Work on BDDs [BRB90] laid the foundation for a new generation of model checkers with the possibility to handle system with large size. The implementations of these new model checkers represent the states of the transition graphs by Boolean formulae which are symbolically represented by BDDs (*cf.* [McM92]).

Finite automata theory provides a formal basis for temporal model checking. From the automata-theoretic point of view, LTL and also CTL model checking can be translated into showing the inclusion of ω-regular languages [Kur89, HK90, VW86]. In the case of LTL model checking, on which we concentrate in this exposition, both the system M as well as the specification formula ϕ are converted into Büchi automata B_M and B_ϕ respectively. The language $L(B_M)$ contains all computations of M and the language $L(B_\phi)$ contains only the allowed computations of M. The formula ϕ is valid over all computations of M iff $L(B_M)$ is a subset of $L(B_\phi)$ or equivalently the intersection $L(B_M) \cap L(\overline{B_\phi})$ is empty. Because Büchi automata are closed under intersection and complementation and their emptiness problem is decidable, the method outlined above represents an approach to LTL model checking based on automata. A well-known LTL model checker based on automata theory is the SPIN system [Hol97].

In this chapter we will provide a formal basis for LTL model checking using the theory of finite automata on finite words. This leads to a new automata-based model checker in which finite automata are used instead of Büchi automata and this offers access to the automata constructions for finite words which are simpler and more efficient as for Büchi automata. Moreover, it allows the use of the automaton minimization operation which is not available for Büchi automata.

Based on the idea of the finite representation of lasso-words introduced in Chapter 4, we give a translation of finite-state systems into M2L-Str formulae and also a translation of LTL formulae into M2L-Str formulae such that if A_M and A_ϕ are the DFAs constructed from the translation of M and ϕ in M2L-Str, then ϕ is valid over all computations of M iff $L(A_M) \subseteq L(A_\phi)$ and thus LTL model checking is reduced to the inclusion of regular languages. We demonstrate how the Mona system can be used as an LTL model checker and show that using this approach we obtain an LTL bounded model checker for free. Namely, we apply the bounded model constructor for M2L-Str to the formula obtained by embedding the underlying model checking problem and a bound k. We will also prove that our LTL bounded model checker and the procedure provided by Biere et al. in [BCCZ99] produce the same (up to variable renaming) Boolean formula.

The remaining of the chapter is organized as follows. In Section 5.2 we briefly introduce finite-state systems, review the linear temporal logic (LTL) and define the model checking problem. In the Sections 5.3.1-5.3.3 we present the M2L-Str encoding of Büchi automata, finite-state systems, and LTL formulae. Furthermore, we show how LTL satisfiability can be decided by using the Mona system and we show how LTL model checking is encoded as a validity problem in M2L-Str and demonstrate how the Mona system can be used as a model checker. In Section 5.3.4 we provide some optimizations of the previous encodings and in Section 5.4, we briefly review the bounded model checking problem and explain how the bounded model constructor for M2L-Str can serve as a bounded model checker.

5.2 LTL Model Checking

Finite-State Systems A *finite-state system* M is given a Kripke structure $M = (S, I, T, L)$ over a set of atomic propositions $\mathcal{P}$, where S is a finite set of states, $I \subseteq S$ is a set of initial states, $T \subseteq S \times S$ is a transition relation, and L is a labeling function of the states $L : S \to 2^{\mathcal{P}}$. The size of M, denoted by $|M|$, is the number of its states. A *path* in M is an infinite sequence of states $s_0 s_1 \ldots$, where the first state is initial, *i.e.* $s_0 \in I$, and for $i \geq 0$, $(s_i, s_{i+1}) \in T$ holds. A *computation* in M is obtained from a path by replacing each state s with the set of atomic propositions labeling it, $L(s)$. With $L(M)$ we denote the set of the computations of M.

Without loss of generality, we assume that any finite-state system $M = (S, I, T, L)$ has a *Boolean encoding* (S', I', T', L'). This means, there are some $m, n \in \mathbb{N}$ such that every state $s \in S$ is encoded by an n-bit vector $\overline{s}$ in $\mathbb{B}^n$ and every subset P of $\mathcal{P}$ is encoded by an m-bit vector $\overline{P} \in \mathbb{B}^m$ and

- $S' = \mathbb{B}^n$,
- $I' : \mathbb{B}^n \to \mathbb{B}$ with $I'(\overline{s})$ holds iff $s \in I$,
- $T' : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$ with $T'(\overline{s_1}, \overline{s_2})$ holds iff $(s_1, s_2) \in T$, and
- $L' : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}$ with $L'(\overline{s}, \overline{P})$ holds iff $L(s) = P$.

Analogously, we assume that any Büchi automaton $B = (S, S_0, \delta, F)$ over an alphabet Σ has a Boolean encoding $B = (S', S'_0, \delta', F)$; that is there are $m, n \in \mathbb{N}$ such that any state $s \in S$ is encoded by an n-bit vector $\overline{s} \in \mathbb{B}^n$ and any symbol $a \in \Sigma$ is encoded by an m-bit vector $\overline{a} \in \mathbb{B}^m$ and

- $S' = \mathbb{B}^n$,
- $S'_0 : \mathbb{B}^n \to \mathbb{B}$ with $S'_0(\overline{s})$ holds iff $s \in S_0$,
- $\delta' : \mathbb{B}^n \times \mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}$ with $\delta'(\overline{s_1}, \overline{a}, \overline{s_2})$ holds iff $s_2 \in \delta(s_1, a)$,
- $F' : \mathbb{B}^n \to \mathbb{B}$ with $F'(\overline{s})$ holds iff $s \in F$.

Unless indicated otherwise, Büchi automata and finite-state systems are for the rest of this chapter given by their Boolean encodings.

The connection between finite-state systems and Büchi automata is straightforward. A finite-state system $M = (S, I, T, L)$ can be viewed as a Büchi automaton

$B = (S, I, \delta, F)$ over $\mathbb{B}^m$, where (i) $\delta(s_1, a, s_2)$ holds iff both $T(s_1, s_2)$ and $L(s_1, a)$ hold and (ii) for all $s \in S$, $F(s)$ holds. The automaton B has the entire set of states S as final states and so any run of B is accepting. Thus, the set of computations of M, $L(M)$, coincides with the set $L(B)$.

Linear Temporal Logic LTL The formulae of LTL are built from a set $\mathcal{P}$ of atomic propositions and are closed under Boolean connectives, the next-time operator next and the until operator U. The modality $\mathsf{next}\,\phi$ means that ϕ holds at the next time point. The modality $\phi \mathsf{U} \psi$ means that there is some time point in the future where the formula ψ holds and the formula ϕ holds everywhere before that point. Note that we adopt here the notion of *reflexive future*; that is, the future includes the present.

LTL formulae are interpreted over computations. These are sequences $\pi : \mathbb{N} \to 2^{\mathcal{P}}$ of sets of atomic propositions. We use the notation π^i to denote the i-shifted computation defined by $\pi^i(j) = \pi(i + j)$, for $j \geq 0$. We will use π and π^0 interchangeably. We define satisfiability of a formula ϕ inductively on its structure as follows:

$$\begin{array}{lll}
\pi \models_{\text{LTL}} p, & \text{if} & p \in \pi(0), \text{ for } p \in \mathcal{P} \\
\pi \models_{\text{LTL}} \neg\phi, & \text{if} & \text{not } \pi \models_{\text{LTL}} \phi \\
\pi \models_{\text{LTL}} \phi_1 \vee \phi_2, & \text{if} & \pi \models_{\text{LTL}} \phi_1 \text{ or } \pi \models_{\text{LTL}} \phi_2 \\
\pi \models_{\text{LTL}} \mathsf{next}\,\phi, & \text{if} & \pi^1 \models_{\text{LTL}} \phi \\
\pi \models_{\text{LTL}} \phi_1 \mathsf{U} \phi_2, & \text{if} & \text{there is some } i \geq 0, \text{ with } \pi^i \models_{\text{LTL}} \phi_2 \text{ and for } 0 \leq j < i \\
 & & \pi^j \models_{\text{LTL}} \phi_1
\end{array}$$

If $\pi \models_{\text{LTL}} \phi$, we say that π satisfies a formula ϕ or also ϕ is valid over the computation π.

Based on the syntax and semantics of LTL described above, we can define other commonly used time operators. First, the truth constant true (respectively, false) stands as an abbreviation for $p \vee \neg p$ (respectively $p \wedge \neg p$), for some proposition p in $\mathcal{P}$. The operator $\Diamond$ is defined by $\Diamond\phi \overset{def}{=} \mathsf{true}\,\mathsf{U}\phi$ and means that ϕ holds some time in the future. The operator $\Box$ is defined by $\Box\phi \overset{def}{=} \neg\Diamond\neg\phi$ and means that ϕ holds everywhere in the future. The operator $\Diamond^\infty$ is defined by $\Diamond^\infty\phi \overset{def}{=} \Box\Diamond\phi$ and means that ϕ holds infinitely often in the future. Finally, the operator $\Box^\infty$ is defined by $\Box^\infty\phi \overset{def}{=} \Diamond\Box\phi$ and means that ϕ holds almost everywhere in the future.

It is known that LTL, FO[<], and star-free ω-regular languages are equiexpressive [Kam68b, Zuc86]. One interesting remark here is that FO[<] is significantly (non-elementary) more succinct than LTL (*cf.* [Mey75]). The connection between LTL and Büchi automata is established in several works [GPVW95, LP85]. An LTL formula ϕ can be translated inductively into a Büchi automaton of size $O(2^{|\phi|})$ that accepts exactly the computations over which ϕ is valid.

LTL Model Checking For a given finite-state system M and an LTL formula ϕ, the model checking problem consists of checking whether all computations of M satisfy ϕ.

Definition 5.2.1
Model Checking for LTL *(*MC(LTL)*)*
INSTANCE: *A finite-state system M and a formula ϕ.*
QUESTION: *Is $\pi \models_{LTL} \phi$ for all computations π in $L(M)$.*

The LTL model checking problem can be reduced to a containment problem of ω-languages as follows. For M and ϕ, we construct the Büchi automata B_M and B_ϕ respectively as we mentioned previously. Then checking whether M is a model of ϕ is equivalent to verifying that all computations of B_M are accepted computations of B_ϕ, which means $L(B_M) \subseteq L(B_\phi)$.

Now, our aim is to go one step further and reduce the LTL model checking problem into a containment problem of regular languages. By Proposition A.1.1, we know that $L(B_M) \subseteq L(B_\phi)$ holds iff all lasso-words from $L(B_M)$ are also in $L(B_\phi)$. In the following we will show how in general a Büchi automaton B can be translated into a DFA A such that from any accepted lasso-word of B, we can extract an accepting word of A and conversely, we can extend any accepted word of A to an accepted lasso-word of B. Let A_M and A_ϕ be the DFAs generated from the Büchi automata B_M and B_ϕ respectively. Then $L(B_M) \subseteq L(B_\phi)$ holds iff $L(A_M) \subseteq L(A_\phi)$ holds too. Our construction of A_M and A_ϕ is indirect. We describe them declaratively as M2L-STR formulae.

5.3 Translating LTL Model Checking in M2L-Str

The LTL model checking problem is translated into the monadic logic M2L-STR as follows: For a finite-state system M and a formula ϕ we translate M into an M2L-STR formula, say $\lceil M \rceil$, and we translate ϕ into an M2L-STR formula, say $\lceil \phi \rceil$, such that M is a model of ϕ iff the implication $\lceil M \rceil \rightarrow \lceil \phi \rceil$ is valid in M2L-STR.

In the following sections we describe how the formulae $\lceil M \rceil$ and $\lceil \phi \rceil$ are constructed.

5.3.1 Encoding of Büchi Automata into M2L-Str

We first show how Büchi automata are translated into M2L-STR and afterwards specialize our encoding to finite-state systems.

Let $B = (S, I, \delta, F)$ be a Büchi automaton over $\mathbb{B}^m$. The encoding of B in M2L-STR is denoted by the formula $\lceil B \rceil$. $\lceil B \rceil$ involves the predicates I, δ, and F

which can be straightforwardly expressed in M2L-Str. Words over $\mathbb{B}^m$ that can be accepted by B are encoded using the variables $X_1, \ldots, X_m$ and the runs of B are encoded using the variables $S_0, \ldots, S_{n-1}$. Words that satisfy $\lceil B \rceil$ are of the form uv, where $u, v \in \mathbb{B}^m$. We use the variable l to encode $|\,u\,|$ and the constant \$ to encode $|\,uv\,|$.

Definition 5.3.1 *Let $B = (S, I, \delta, F)$ be a Büchi automaton over $\mathbb{B}^m$. We define the* M2L-Str *encoding $\lceil B \rceil$ of B as follows:*

$$\begin{aligned}
\lceil B \rceil = {} & \exists l.\, \exists S_0, \ldots, S_{n-1}. \\
& I(S_0(0), \ldots, S_{n-1}(0)) \wedge && (5.1)\\
& \forall p.\, p > 0 \to \\
& \quad \delta(S_0(p-1), \ldots, S_{n-1}(p-1), X_1(p), \ldots, X_{m-1}(p), S_0(p), \ldots, S_{n-1}(p)) \wedge && (5.2)\\
& \delta(S_0(\$), \ldots, S_{n-1}(\$), X_1(\$), \ldots, X_{m-1}(\$), S_0(l), \ldots, S_{n-1}(l)) \wedge && (5.3)\\
& \exists p.\, l \leq p \wedge F(S_0(p), \ldots, S_{n-1}(p))\,. && (5.4)
\end{aligned}$$

A word w satisfies $\lceil B \rceil$ iff there is a run which satisfies the following conditions: it starts with an initial state (5.1), every two consecutive states obey the transition relation (5.2), its last state has a back loop to some of the previous states (5.3), and this loop comprises a final state (5.4).

Theorem 5.3.2 *Let B be a Büchi automaton over $\mathbb{B}^m$ and A be the* DFA *representing $\lceil B \rceil$. The following holds:*

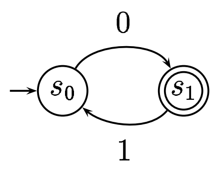
(1) If $w \in L(A)$ then there are $u, v \in (\mathbb{B}^m)^*$ with $w = uv$ and $uv^\omega \in L(B)$.

(2) If $uv^\omega \in L(B)$ then there is an $n \in \mathbb{N}$ such that $uv^n \in L(A)$.

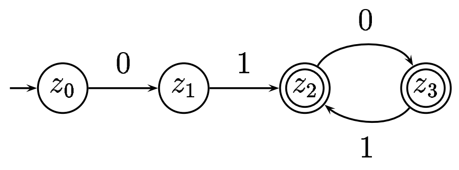
Proof (1): follows immediately by the construction of $\lceil B \rceil$.
(2): Let u and v be in Σ^* with $uv^\omega \in L(B)$. Let r be an accepting run $r : \mathbb{N} \to S$ of uv^ω. Consider the infinite sequence $r(|u|)$, $r(|u| + |v|)$, ..., $r(|u| + p|v|)$, ... There exist i, j, and k with $i \leq j < k$ and such that $r(|u| + i|v|) = r(|u| + k|v|)$ and $r(|u| + j|v|)$ is a final state. By choosing $n = k - i$, the run $r(0)$, $r(1)$, ..., $r(|u| + n|v|)$ and the word uv^n satisfy the formula $\lceil B \rceil$ and hence $uv^n \in L(A)$. ■

Notice that in the above theorem the n repetitions of v is necessary in order to obtain a loop containing a final state.

Example 5.3.1 To illustrate the encoding of Büchi automata, we give the following example. Let B be the Büchi automaton accepting the language containing just the word $(01)^\omega$. B is visualized as follows.



The DFA A corresponding to the encoding $\lceil B \rceil$ is depicted as follows.

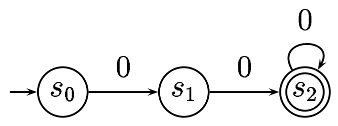


The language accepted by A is $(01)^+ \cup 0(10)^+$. Any accepted word of A can be extend to the lasso-word $(01)^\omega$: let $n > 0$. A word of the form $(01)^n$ in $L(A)$ can be extended to the lasso-word $uv^\omega = (01)^\omega$, where u is the empty word and $v = (01)^n$ and a word of the form $0(10)^n$ in $L(A)$ can be extended to the lasso-word $uv^\omega = (01)^\omega$, where $u = 0$ and $v = (10)^n$. Conversely, for all words $u, v \in \{0,1\}^*$ if $uv^\omega = (01)^\omega$, then uv is accepted by A. □

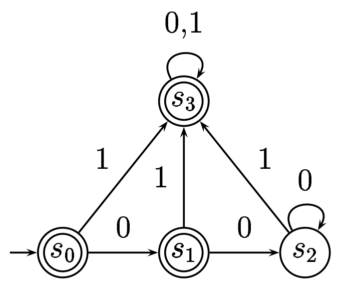
By Theorem 5.3.2, we can easily deduce that the emptiness problem of Büchi automata can be translated into the emptiness problem of DFAs.

Theorem 5.3.3 *Let B be a Büchi automaton and A be a* DFA *representing $\lceil B \rceil$, the* M2L-Str *encoding of B. $L(B)$ is empty iff $L(A)$ is empty.*

Note that the encoding $\lceil . \rceil$ of Büchi automata is not closed under the union $\cup$, intersection $\cap$, and complementation $^-$ operations. For example, the encoding of the complement of a Büchi automaton B, $\lceil \overline{B} \rceil$, and the negation of the encoding of B, $\neg \lceil B \rceil$, are not equivalent M2L-Str formulae. To show this, consider the following Büchi automaton B which accepts the unique infinite word 0^ω.



The DFA A representing the encoding of B has the same graph as B and its complement $\overline{A}$ is the following DFA.



We can see that $0 \in L(\overline{A})$, but 0 can only be extended to 0^ω, which is in $L(B)$ and not in $L(\overline{B})$.

Observe that, since $\lceil . \rceil$ is not closed under the operations $\cup$, $\cap$, and $\bar{}$, we can not use our approach for compositional reasoning. Assume that we are given a system specified as the intersection of two Büchi automata and we want to check some property of the system. To achieve this, we encode both the system and its property in M2L-STR as a validity problem. An improvement of this method would consists of decomposing the system into its ("simpler") components and then prove the property for each of the components separately. This is, however, not correct because, as we mention above, $\lceil . \rceil$ is not closed under intersection.

Independently from our work, Mödersheim [Möd01] gave an encoding of Büchi automata into WS1S. He proved that his encoding is closed under union and intersection, but not under complementation. Furthermore, he showed how his encoding can be used for verifying non-terminating systems where both models and the negation of properties to be checked are specified as Büchi automata.

5.3.2 Encoding of Finite-state Systems into M2L-Str

As mentioned before, finite-state systems are Büchi automata where the entire set of states are final states. This restriction on Büchi automata leads to the essential property that the encoding of finite-state systems in M2L-STR becomes closed under union, intersection, and complementation.

Definition 5.3.4 *Let $M = (S, I, T, L)$ be a finite-state system. We define $\lceil M \rceil$ as follows.*

$$\begin{aligned}
\lceil M \rceil = {} & \exists l.\, \exists S_0, \ldots, S_{n-1}. && (5.5)\\
& I(S_0(0), \ldots, S_{n-1}(0)) \wedge && (5.6)\\
& \forall p.\, L(S_0(p), \ldots, S_{n-1}(p), X_1(p), \ldots, X_{m-1}(p)) \wedge && (5.7)\\
& \forall p.\, p > 0 \to T(S_0(p-1), \ldots, S_{n-1}(p-1), S_0(p), \ldots, S_{n-1}(p)) \wedge && (5.8)\\
& T(S_0(\$), \ldots, S_{n-1}(\$), S_0(l), \ldots, S_{n-1}(l))\,. && (5.9)
\end{aligned}$$

With $\lceil M \rceil_l$ we denote the formula obtained from $\lceil M \rceil$ by removing the $\exists$-quantifier of the variable l in (5.5).

We can adapt Theorem 5.3.2, for finite-state systems as follows.

Theorem 5.3.5 *Let M be a finite-state system and A a* DFA *representing $\lceil M \rceil$.*

(1) If $w \in L(A)$ then there are $u, v \in \Sigma^*$ with $w = uv$ and $uv^\omega \in L(M)$.

(2) If $uv^\omega \in L(M)$ then $uv \in L(A)$.

In Theorem 5.3.5(2) the n repetitions of v are in order to obtain a loop containing a final state are not necessary, because all states in M are final. This observation is the main reason why the encoding of finite-state systems is closed under union, intersection, and complementation.

Theorem 5.3.6 *Let M, M_1, and M_2 be finite-state systems. We have:*

$$\begin{aligned}\lceil M_1 \cup M_2 \rceil &= \lceil M_1 \rceil \vee \lceil M_2 \rceil \\ \lceil M_1 \cap M_2 \rceil &= \lceil M_1 \rceil \wedge \lceil M_2 \rceil \\ \lceil \overline{M} \rceil &= \neg \lceil M \rceil\end{aligned}$$

Proof We only prove the closure property for the complementation; the proof of the others is straightforward.

Let A_1 and A_2 be the DFAs constructed from $\lceil M \rceil$ and $\lceil \overline{M} \rceil$. Our goal is to prove that A_2 is the complement of A_1; that is, for all $w \in \Sigma^*$, $w \notin L(A_1)$ iff $w \in L(A_2)$.

Assume $w \in L(A_1)$ and $w \in L(A_2)$. From $w \in L(A_2)$ follows, by Theorem 5.3.5(1), that there are $u, v \in \Sigma^*$ with $w = uv$ and $uv^\omega \in L(\overline{M})$. Because all states in M are accepting, there is a position in the word $w = uv$ from which no transition in M is possible. We call this a *stuck* position. From $w \in L(A_1)$ follows, by Theorem 5.3.5(1), that there are $u', v' \in \Sigma^*$ with $w = u'v'$ and $u'(v'^\omega) \in L(M)$. In particular, there is no stuck position in $w = uv = u'v'$ in M, which contradicts what we just stated before.

Now, assume that $w \notin L(A_1)$ and $w \notin L(A_2)$. From these two assumptions and by Theorem 5.3.5(2) we deduce the following two contradictory facts respectively.

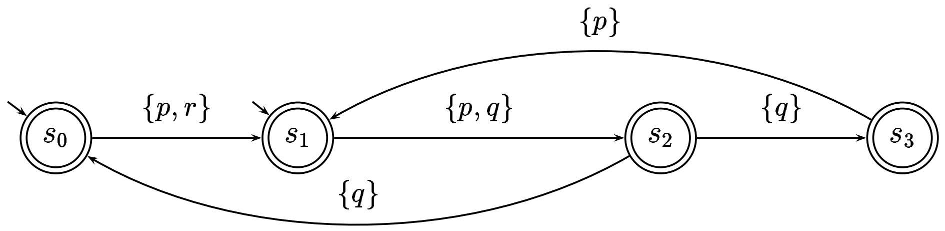
$$\text{for all } u, v \in \Sigma^*, \text{ if } w = uv \text{ then } uv^\omega \notin L(M), \text{ and}$$

$$\text{for all } u, v \in \Sigma^*, \text{ if } w = uv \text{ then } uv^\omega \notin L(\overline{M}).$$

■

Example 5.3.2 The example that we consider here will be used as a running example for the rest of this chapter. We specify a finite-state system, sketch its formulation as a Büchi automaton, provide a Boolean encoding of it, and describe the DFA representing its encoding in M2L-STR.

Let Σ be the alphabet set $\mathsf{Pow}(\{p, q, r\})$ and M be the finite-state system (S, I, T, L), where $S = \{s_0, s_1, s_2, s_3\}$ is the set of states, $I = \{s_0, s_1\}$ is the set of initial states, $T = \{(s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_1), (s_2, s_0)\}$ is the transition relation, and the labeling function L is defined as follows: $L(s_0) = \{p, r\}$, $L(s_1) = \{p, q\}$, $L(s_2) = \{q\}$, and $L(s_3) = \{p\}$. The Büchi automaton B corresponding to M is depicted below:



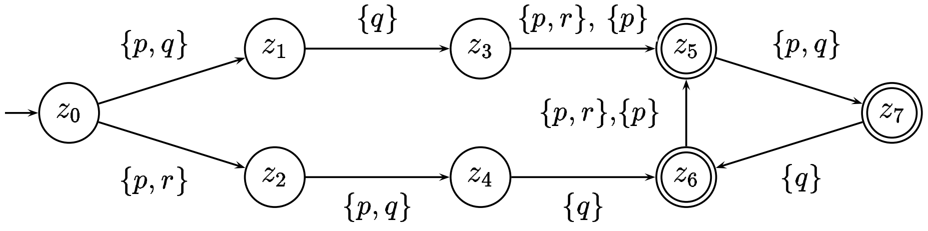
The finite-state system M has the following Boolean encoding (S', I', T', L'): $S' = \mathbb{B}^2$, where the states are encoded as follows: $s_0 = (0, 0)$, $s_1 = (0, 1)$, $s_2 = (1, 0)$, and $s_3 = (1, 1)$. The initial state predicate is I' defined by $I'(x_0, x_1) = \overline{x_0}$, and the transition relation is given by:

$$\begin{array}{rcccccccccc}
T'(x_0, x_1, x_0', x_1') & = & \neg x_0 & \wedge & \neg x_1 & \wedge & \neg x_0' & \wedge & x_1' & & \vee \\
 & & \neg x_0 & \wedge & x_1 & \wedge & x_0' & \wedge & \neg x_1' & & \vee \\
 & & x_0 & \wedge & \neg x_1 & \wedge & \neg x_0' & \wedge & \neg x_1' & & \vee \\
 & & x_0 & \wedge & \neg x_1 & \wedge & x_0' & \wedge & x_1' & & \vee \\
 & & x_0 & \wedge & x_1 & \wedge & \neg x_0' & \wedge & x_1', & &
\end{array}$$

and finally the labeling function is specified by the following predicates:

$$\begin{array}{rcccccccccc}
L(x_0, x_1, p, q, r) & = & \neg x_0 & \wedge & \neg x_1 & \wedge & p & \wedge & \neg q & \wedge & r & \vee \\
 & & \neg x_0 & \wedge & x_1 & \wedge & p & \wedge & q & \wedge & \neg r & \vee \\
 & & x_0 & \wedge & \neg x_1 & \wedge & \neg p & \wedge & q & \wedge & \neg r & \vee \\
 & & x_0 & \wedge & x_1 & \wedge & p & \wedge & \neg q & \wedge & \neg r\,. &
\end{array}$$

The finite automaton A corresponding to $\lceil M \rceil$ is defined as follows:



□

5.3.3 Encoding of LTL into M2L-Str

Let us now turn our attention to the encoding of LTL in M2L-Str. We will give firstly an encoding that is satisfiability preserving and afterwards we slightly modify this encoding for model checking purposes. As we will see, the key idea behind the encoding is again the finite representation of lasso computations in M2L-Str. We want to point out that this encoding maps LTL into the first-order fragment of M2L-Str, FO-Str[<], whereas independently from our work, Hirsch and Hustadt

[HH01] presented an encoding of LTL into WS1S, that involves quantification over first-order and second-order variables.

Intuitively, the encode $\lceil . \rceil$ formalizes in M2L-STR "what it means, that an LTL formula ϕ is valid over a lasso computation". $\lceil \phi \rceil$ is satisfiable if there exists a word, say π, of length \$, which has a back loop from the last position \$ to some previous position l and such that the formula $\lceil \phi \rceil_0^{l,\$}$ holds over π. The mapping $\lceil . \rceil_i^{l,k}$ formalizes the validity of ϕ over a (l,k)-lasso computation starting from an arbitrary position i of the computation.

Let i, l, and k be first-order variables in M2L-STR and ϕ an LTL formula. We associate to every LTL atomic proposition p occurring in ϕ a second-order variable X_p.

Definition 5.3.7 *We define $\lceil \phi \rceil = \exists\, l.\ \lceil \phi \rceil_0^{l,\$}$, where the mapping $\lceil . \rceil_i^{l,k}$ is defined as follows:*[1]

$$\begin{aligned}
\lceil p \rceil_i^{l,k} &= X_p(i) && (5.10)\\
\lceil \phi_1 \vee \phi_2 \rceil_i^{l,k} &= \lceil \phi_1 \rceil_i^{l,k} \vee \lceil \phi_2 \rceil_i^{l,k} &&\\
\lceil \neg \phi \rceil_i^{l,k} &= \neg \lceil \phi \rceil_i^{l,k} &&\\
\lceil \mathsf{next}\, \phi \rceil_i^{l,k} &= \mathsf{if}\ i < k-1\ \mathsf{then}\ \lceil \phi \rceil_{i+1}^{l,k}\ \mathsf{else}\ \lceil \phi \rceil_l^{l,k} &&\\
\lceil \phi_1 \mathsf{U} \phi_2 \rceil_i^{l,k} &= \exists p.\, p \leq k\ \wedge \lceil \phi_2 \rceil_p^{l,k} \wedge && (5.11)\\
&\qquad \mathsf{if}\ i \leq p\ \mathsf{then}\ \forall x.\, i \leq x < p \rightarrow \lceil \phi_1 \rceil_x^{l,k} && (5.12)\\
&\qquad \mathsf{else}\ l \leq p \wedge (\forall x.\, (l \leq x < p \vee i \leq x \leq k) \rightarrow \lceil \phi_1 \rceil_x^{l,k}) && (5.13)
\end{aligned}$$

From the above definition, we can deduce the following facts.

$$\begin{aligned}
\lceil \mathsf{true} \rceil_i^{l,k} &= \mathsf{true}\\
\lceil \Diamond \phi \rceil_i^{l,k} &= \exists j.\, \mathsf{min}(i,l) \leq j \leq k\ \wedge \lceil \phi \rceil_j^{l,k}\\
\lceil \Box \phi \rceil_i^{l,k} &= \forall j.\, \mathsf{min}(i,l) \leq j \leq k\ \rightarrow \lceil \phi \rceil_j^{l,k}\\
\lceil \Diamond^\infty \phi \rceil_i^{l,k} &= \forall j.\, \mathsf{min}(i,l) \leq j \leq k\ \rightarrow \exists p.\, \mathsf{min}(j,l) \leq p \leq k\ \wedge \lceil \phi \rceil_p^{l,k}\\
\lceil \Box^\infty \phi \rceil_i^{l,k} &= \exists j.\, \mathsf{min}(i,l) \leq j \leq k\ \wedge \forall p.\, \mathsf{min}(j,l) \leq p \leq k\ \rightarrow \lceil \phi \rceil_p^{l,k}
\end{aligned}$$

[1]The formula $\mathsf{if}\ A\ \mathsf{then}\ B\ \mathsf{else}\ C$ is an abbreviation for the formula $(A \rightarrow B) \wedge (\neg A \rightarrow C)$.

Le p be an LTL atomic proposition. The following facts hold.

$$\begin{array}{rcl}\lceil \Diamond p \rceil & = & \exists j.\, X_p(j) \\ \lceil \Box p \rceil & = & \forall j.\, X_p(j) \\ \lceil \Diamond^\infty p \rceil & = & \exists l.\, \forall i.\, \exists j.\, (i \leq j \vee l \leq j) \wedge X_p(j) \\ \lceil \Box^\infty p \rceil & = & \exists l.\, \exists i.\, \forall j.\, \mathsf{min}(i,l) \leq j \rightarrow X_p(j)\end{array}$$

LTL formulae are interpreted over computations, which are infinite sequences of sets of atomic propositions. To establish the correctness of the translation of LTL formulae into M2L-Str, we need to define how computations can be seen as M2L-Str substitutions.

Definition 5.3.8 *Let π be a computation. We define with $\lceil \pi \rceil$ the* M2L-Str *substitution given by $\lceil \pi \rceil(X_p) = \{i \mid p \in \pi(i)\}$.*

Fact 5.3.9 *If π is a (l,k)-lasso computation, then $\lceil \pi \rceil$ is a (l,k)-lasso substitution.*

Lemma 5.3.10 *Let π be an (l,k)-lasso computation and ϕ be an* LTL *formula. Then $\pi^i \models_{LTL} \phi$ holds iff $\lceil \pi \rceil^k \models_{M2L} \lceil \phi \rceil^{l,k}_{i'}$ holds, for all $i \geq 0$ and $i' = \langle i \rangle^l_k$.*

Proof We proceed by induction over the structure of the formula ϕ. We begin by observing that, by Fact 5.3.9, $\lceil \pi \rceil$ is a (l,k)-lasso substitution.

Let us consider the case where ϕ is an atomic proposition p. The claim is to prove that

$$\pi^i \models_{\text{LTL}} p \text{ iff } \lceil \pi \rceil^k \models_{\text{M2L}} X_p(\langle i \rangle^l_k), \text{ for all } i \geq 0.$$

By the semantics of LTL, $\pi^i \models_{\text{LTL}} p$ is equivalent to $p \in \pi(i)$. The claim follows now because $p \in \pi(i)$ and $i \in \lceil \pi \rceil(X_p)$ are equivalent and by Lemma B.1.1, $i \in \lceil \pi \rceil(X_p)$ is equivalent to $\langle i \rangle^l_k \in \lceil \pi \rceil(X_p)$.

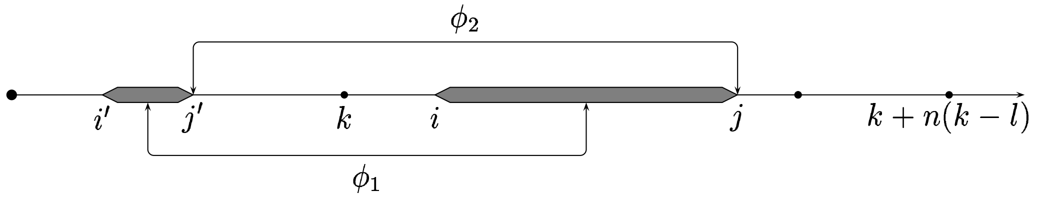
In the induction step we consider first the case where ϕ is of the form $\phi_1 \mathsf{U} \phi_2$ and we prove the direction: $\pi^i \models_{\text{LTL}} \phi_1$ implies $\lceil \pi \rceil^k \models_{\text{M2L}} \lceil \phi \rceil^{l,k}_{i'}$, where $i' = \langle i \rangle^l_k$. By the semantics of LTL, $\pi^i \models_{\text{LTL}} \phi_1 \mathsf{U} \phi_2$ holds iff there is some $i \leq j$ with $\pi^j \models_{\text{LTL}} \phi_2$ and $\pi^m \models_{\text{LTL}} \phi_1$ for $i \leq m < j$. Let $i' = \langle i \rangle^l_k$, $j' = \langle j \rangle^l_k$, and $m' = \langle m \rangle^l_k$. By the induction hypothesis we obtain

$$\lceil \pi \rceil^k \models_{\text{M2L}} \lceil \phi_2 \rceil^{l,k}_{j'}, \text{ and} \tag{5.14}$$

$$\lceil \pi \rceil^k \models_{\text{M2L}} \lceil \phi_1 \rceil^{l,k}_{m'}\,. \tag{5.15}$$

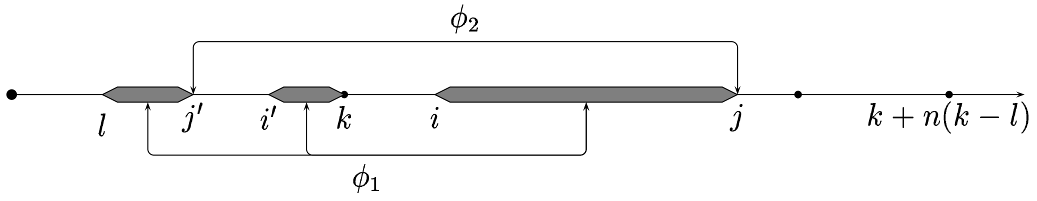
We distinguish two cases:

- $i' \leq j'$: The situation can be visualized as follows.



For each x with $i' \leq x < j'$ there is some m with $i \leq m < j$ such that $\langle m \rangle_k^l = x$. By (5.15), we conclude that $\lceil \pi \rceil^k \models_{\text{M2L}} \forall x.\, i' \leq x < j' \to \lceil \phi_1 \rceil_x^{l,k}$ holds. Thus by choosing $p = j'$ in (5.11) and by (5.14), it follows that $\lceil \pi \rceil^k \models_{\text{M2L}} \lceil \phi_1 \mathsf{U} \phi_2 \rceil_{i'}^{l,k}$ holds.

- $j' < i'$: From the fact that $j' < i'$ it follows first that $l \leq j'$. The situation is again visualized in the following picture.



For each x with $l \leq x < j' \vee i' \leq x < k$ there is some m with $i \leq m < j$ such that $\langle m \rangle_k^l = x$. By (5.15), we conclude that $\lceil \pi \rceil^k \models_{\text{M2L}} \forall x.\, l \leq x < j' \vee i' \leq x < k \to \lceil \phi_1 \rceil_x^{l,k}$ holds. Thus by choosing $p = j'$ in (5.11) and by (5.14), it follows that $\lceil \pi \rceil^k \models_{\text{M2L}} \lceil \phi_1 \mathsf{U} \phi_2 \rceil_{i'}^{l,k}$ holds too.

Conversely, we can similarly establish that for $i' = \langle i \rangle_k^l$, $\lceil \pi \rceil^k \models_{\text{M2L}} \lceil \phi_1 \mathsf{U} \phi_2 \rceil_{i'}^{l,k}$ implies $\pi^i \models_{\text{LTL}} \phi_1 \mathsf{U} \phi_2$.

Let us consider the case where ϕ is of the form $\mathsf{next}\, \phi'$. By the semantics of LTL, $\pi^i \models_{\text{LTL}} \mathsf{next}\, \phi'$ and $\pi^{i+1} \models_{\text{LTL}} \phi'$ are equivalent. By the induction hypothesis, we conclude that $\pi^{i+1} \models_{\text{LTL}} \phi'$ and $\lceil \pi \rceil^k \models_{\text{M2L}} \lceil \phi \rceil_j^{l,k}$, for $j = \langle i+1 \rangle_k^l$ are equivalent. Now, (3) and (4) from Lemma B.1.1 conclude the claim.

The remaining cases where ϕ is of the form $\phi_1 \vee \phi_2$ or $\neg \phi'$ are straightforward. ■

We now reformulate Lemma 5.3.10 as follows.

Lemma 5.3.11 *Let π be an (l,k)-lasso computation and let ϕ be an* LTL *formula. Then $\pi \models_{LTL} \phi$ iff $\lceil \pi \rceil^k \models_{M2L} \lceil \phi \rceil$.*

Theorem 5.3.12 (LTL Satisfiability in M2L-Str) *An* LTL *formula ϕ is satisfiable iff $\lceil \phi \rceil$ is satisfiable in* M2L-Str.

Proof We know by Proposition 4.3.2 that ϕ is satisfiable iff there is an (l,k)-lasso computation π over which ϕ is valid. By Lemma 5.3.11, it follows that $\lceil \pi \rceil^k \models_{\text{LTL}} \lceil \phi \rceil$ holds too. ■

The above Lemma builds the basis of an automata-based decision procedure for LTL. For example, the MONA system can be utilized for this task. MONA invoked with the encoding of an LTL formula calculates a deterministic automaton whose accepted words correspond to the computations over which the given LTL formula is valid.

Now, we show that also the model checking problem can be formalized in M2L-Str.

Definition 5.3.13 *Let M be a finite-state system and ϕ be an* LTL *formula. We define the mapping $[\![.,.]\!]$ by $[\![M,\phi]\!] = \forall l.\ \lceil M \rceil_l \rightarrow \lceil \phi \rceil_0^{l,\$}$.*

Theorem 5.3.14 (LTL Model Checking in M2L-Str) *Let M be a finite-state system and ϕ be an* LTL *formula. M is a model of ϕ iff $[\![M,\phi]\!]$ is valid in* M2L-Str.

Proof " $\Rightarrow$ ": assume that M is a model of ϕ and let $\sigma = [X_{p_1}/E_1, \ldots, X_{p_m}/E_m, l/c]$ be a substitution that satisfies $\lceil M \rceil_l$. We recall that the variables X_{p_1}, …, X_{p_m} encode the propositions p_1, …, p_m respectively. The variable l encodes the position where the loop starts and c is a natural number. Let the word uv over $\mathbb{B}^m$ corresponding to the substitution $[X_{p_1}/E_1, \ldots, X_{p_m}/E_m]$ such that $|u| = c$. It obviously follows that $uv^\omega \in L(M)$ and by assumption we conclude that ϕ is valid over uv^ω. Thus, by Lemma 5.3.10, σ satisfies $\lceil \phi \rceil_0^{l,\$}$.

" $\Leftarrow$ ": assume $\forall l.\ \lceil M \rceil_l \rightarrow \lceil \phi \rceil_0^{l,\$}$ is satisfied in M2L-Str and $uv^\omega \in L(M)$. By Theorem 5.3.5, the substitution σ obtained from uv^ω satisfies $\lceil M \rceil_l$, if we additionally interpret the variable l by $|u|$. By assumption, we obtain that $\sigma[l/\ |u|]$ satisfies $\lceil \phi \rceil_0^{l,\$}$. Now, by Lemma 5.3.10, we conclude that ϕ is valid over uv^ω. ■

The M2L-Str decision procedure implemented in the MONA system can be used as an LTL model checker. Given an instance (M, ϕ), we first compute the formula $[\![M,\phi]\!]$ and then call MONA. When MONA succeeds to calculate the DFA corresponding to $[\![M,\phi]\!]$, it responds either with "valid" indicating that M is effectively a model of ϕ or it produces a counter-example that can be used for debugging purposes.

Example 5.3.3 (continued) We checked three properties of the finite-state system M using MONA. The first property is $\Diamond p$ and means that the atomic proposition p occurs in every computation of M. MONA invoked with the formula $[\![M, \Diamond p]\!]$ responds with:

```
Formula is valid
Total time:  00:00:00.05
```

The second property is $\Diamond r$ and MONA invoked with the formula $[\![M, \Diamond r]\!]$ produces a counter-example in this case.

```
A counter-example of least length (3) is:
P 101
Q 110
R 000
Total time:  00:00:00.05
```

The decoding of the counter-example yields the finite computation $\{p, q\}\{q\}\{p\}$.

The third property is $\Diamond r \to \Diamond^{\infty} r$ and means that whenever r occurs in a computation of M then r occurs infinitely often in that computation. MONA also produces a counter-example for $[\![M, \Diamond r \to \Box^{\infty} r]\!]$.

```
A counter-example of least length (4) is:
P 1101
Q 0110
R 1000
Total time:  00:00:00.08
```

The decoding of the counter-example yields the finite computation $\{p, r\}\{p, q\}\{q\}\{p\}$.

The verification of the two last properties points to a weakness of using MONA as an LTL model checker. When MONA calculates a counter-example, we have still to determine how this counter-example should be extended into a lasso computation of the finite-state system under consideration. The obtained computation serves to debug the system for the errors causing the counter-example. If we reconsider the above example where we checked whether $M \models \Diamond r$ holds, we will notice that there are three possibilities to extend the counter-example $\{p, q\}\{q\}\{p\}$ to a lasso computation; for example, $(\{p, q\}\{q\}\{p\})^{\omega}$, $\{p, q\}(\{q\}\{p\})^{\omega}$ and $\{p, q\}\{q\}\{p\}^{\omega}$ all are possible extensions and only the first extension is a computation of M. The missing information that we need here to correctly and uniquely determine the lasso computation is the position where the loop should start. This position is interpreted with the variable l in the definition of $[\![., .]\!]$ (Definition 5.3.13). Thus, one way to overcome this problem consists of dropping the $\forall$-quantification over the variable l there; this minor modification has no effect on the results proved so far.

If we change the MONA script as proposed above, MONA provides us with more information: a finite portion of a computation and the position, the value of the variable `l`, where the loop starts.

```
A counter-example of least length (3) is:
l = 0
P 101
Q 110
```

```
R 000
Total time:  00:00:00.08
```

Knowing the value 0 of `l` makes the extension of the counter-example $\{p,q\}\{q\}\{p\}$ to the lasso computation $(\{p,q\}\{q\}\{p\})^\omega$ unique. □

5.3.4 Optimization

In the example before, we demonstrated how LTL model checking can be achieved by the MONA system. Here, we show how we can modify the encodings of finite-state systems as well as of LTL formulae in order to improve the amount of time and space needed by MONA to perform LTL model checking.

By considering the encodings of finite-state systems and LTL formulae, we notice that both encodings are M2L-Str formulae describing regular sets of computations (words over subsets of $\mathcal{P}$). The formula $\lceil M \rceil$ characterizes the set of finite computations that can be extended to lasso computations in M and the formula $\lceil \phi \rceil$, for an LTL formula ϕ, characterizes finite computations that can be extended to lasso computations over which ϕ is valid. We can slightly modify our two encodings such that they describe paths (sequences of states of M) instead of computations.

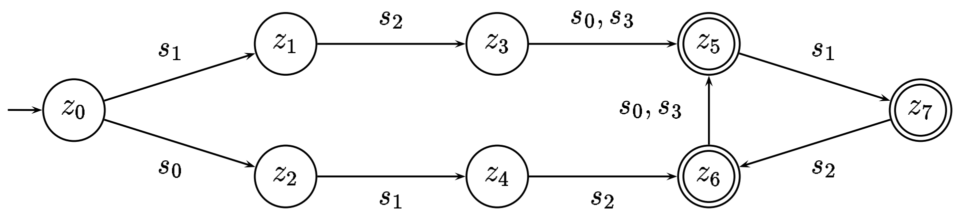
The encoding of a finite-state system M becomes the following M2L-Str formula.

$$\begin{aligned}
[\![M]\!] \;=\; & \exists l.\, I(S_0(0),\dots,S_{n-1}(0)) \wedge \\
& \quad \forall p.\, p > 0 \to T(S_0(p-1),\dots,S_{n-1}(p-1),S_0(p),\dots,S_{n-1}(p)) \wedge \\
& \quad T(S_0(\$),\dots,S_{n-1}(\$),S_0(l),\dots,S_{n-1}(l))\,.
\end{aligned}$$

With $[\![M]\!]_l$ we denote the formula obtained from $[\![M]\!]$ by dropping the $\exists$-quantifier of the variable l.

Compared to the previous definition of $\lceil M \rceil$ given in Definition 5.3.4, here we have discarded the formula given in line (5.7) in Definition 5.3.4 that states the relationship between paths and computations in M. We also have dropped the $\exists$-quantifier of the state variables $S_0, \dots, S_{n-1}$.

The DFA A' representing the encoding $[\![M]\!]$, where M is the finite-state system of Example 5.3.2 is given below.



The automaton A' differs from the automaton A from Example 5.3.2 in the labeling of the transitions. In A, the transitions are labeled with subsets of $\mathcal{P}$, whereas in A' they are labeled with states in M. Later, we will explain why it is more efficient to compute A' than A. Now, we want to establish the relation between the regular languages defined by $\lceil M \rceil$ and $[\![M]\!]$.

Let m and n be two natural numbers and M be a finite-state system with the Boolean encoding (S, I, T, L) and $\tau : \mathbb{B}^n \to \mathsf{Pow}(\mathbb{B}^m)$ be the function defined by

$$\tau(s_0, \dots, s_{n-1}) = \{(p_0, \dots, p_{m-1}) \mid L(s_0, \dots, s_{n-1}, p_0, \dots, p_{m-1})\}$$

We extend the mapping τ to words over $\mathbb{B}^n$ as expected. For the finite-state system M from the Example 5.3.2, we have for example,

$$\begin{array}{rcl} \tau(s_1 s_2 s_3) & = & \tau(\begin{pmatrix}0 & 1\end{pmatrix}\begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}1 & 1\end{pmatrix}) \\ & = & \{\begin{pmatrix}1 & 1 & 0\end{pmatrix}\}\{\begin{pmatrix}0 & 1 & 0\end{pmatrix}\}\{\begin{pmatrix}1 & 0 & 0\end{pmatrix}\} \\ & = & \{\{p, q\}\{q\}\{p\}\} \end{array}$$

The relationship between $\lceil M \rceil$ and $[\![M]\!]$ is stated below.

Lemma 5.3.15 $L(\lceil M \rceil) = \tau(L([\![M]\!]))$.

Now, let us compare the effort spent by MONA to build the DFA, say A_1, from the formula $\lceil M \rceil$ and the DFA, say A_2, from the formula $[\![M]\!]$. In the first case, MONA generates four intermediate automata: automaton for every formula in the lines (5.6-5.9); then it builds the product of these four automaton and finally performs a projection of the variables $S_0, \dots, S_{n-1}$. In the second case (for A_2), MONA generates only three of the four automata and it spares the projection of the variables $S_0, \dots, S_{n-1}$. Note that if we additionally remove the $\exists$-quantification over the variable l in $[\![M]\!]$, MONA additionally spares the projection of the variable l.

After having modified the encoding of finite-state systems, we must also modify the encoding of LTL in M2L-STR such that it also characterizes paths instead of computations. This is necessary in order to obtain "comparable" formulae; that is, from the semantics point of view, formulae that define regular languages over the same alphabet and, and from the syntax point of view, formulae that involve the same free variables $S_0, \dots, S_{n-1}$. In Definition 5.3.7, we modify the rule $\lceil p \rceil_i^{l,k} = X_p(i)$ in the following way.

$$\lceil p \rceil_i^{l,k} = L_p(S_0(i), \dots, S_{n-1}(i)),$$

where $L_p(S_0(i), \dots, S_{n-1}(i))$ is a predicate defined in M2L-STR and holds iff p labels the state encoded by the vector $(S_0(i), \dots, S_{n-1}(i))$. We denote by $[\![\phi]\!]$ the new encoding of ϕ. The relationship between $\lceil \phi \rceil$ and $[\![\phi]\!]$ is state as follows.

Lemma 5.3.16 *$w \in L(\lceil\phi\rceil)$ iff for all x with $\tau(x) = w$, $x \in L(\llbracket\phi\rrbracket)$.*

By Lemma 5.3.15 and Lemma 5.3.16, we conclude

Theorem 5.3.17 *$\lceil M\rceil \to \lceil\phi\rceil$ and $\llbracket M\rrbracket \to \llbracket\phi\rrbracket$ define in* M2L-Str*, up to τ, the same regular language. That is, for all words $w \in \mathbb{B}^m$, $w \in L(\lceil M\rceil \to \lceil\phi\rceil)$ iff there is a word x such that $\tau(x) = w$ and $x \in L(\llbracket M\rrbracket \to \llbracket\phi\rrbracket)$.*

Based on the previous theorem, we redefine the encoding of LTL model checking as follows: $[\![M, \phi]\!] = \forall l.\ \llbracket M\rrbracket_l \to \llbracket\phi\rrbracket_0^{l,\$}$ and we refer to this definition when we use $[\![M, \phi]\!]$ in the rest of this chapter.

5.4 Bounded Model Checking for LTL

The highly automation of model checking and the multiple success stories where model checking was used contribute to a wide and increased acceptance of this *push-button* verification approach in the industry. Not surprisingly though, many large systems cannot be verified due to state-space explosion; that is, the number of states of the transition graph grows exponentially by the number of the components building the finite-state system. Several works, like may be abstraction [CGL94], symmetry [CJEF96, ES93, ID93], and partial-order reduction [GP93, Pel94] addressed this problem with respectable success. Lately, a new model checking technique, bounded model checking, was proposed [BCCZ99] and it has shown promising results.

The bounded model checking problem for LTL is defined similarly to BMC and consists of examining if all computations bounded by a fixed length of a given finite-state system, instead of all (unbounded) computations as it is the case in model checking, satisfy a given LTL property.

Definition 5.4.1
Bounded Model Checking for LTL *(*BMCh(LTL)*)*
Instance: *A finite-state system M, a formula ϕ and a natural number k.*
Parameter: *k.*
Question: *Does ϕ have a satisfying lasso computation in M of length k?*

Biere et al. showed in [BCCZ99] that BMCh(LTL) is *NP-complete*. They provide a procedure which translates an BMCh(LTL)-instance (M, ϕ, k) into a conjunction of two Boolean formulae $f \wedge g$ that is satisfiable iff there is a computation of M of length k over which ϕ is not valid. The formula f makes use of a sequence of vectors of state variables $s_0 = (s_0^0, \ldots, s_0^{n-1})$, …,$s_{k-1}$ to characterize lasso paths of M of length k. f is simply generated by unfolding the transition of M k times and imposing that the first vector state s_0 satisfies the initial state predicate of M. The second formula g constrains the paths described by f to those that do not satisfy ϕ. Here we restrict ourself to this brief description and refer to [BCCZ99] for the full story.

In the previous sections, we showed that the LTL model checking can be specified as an M2L-STR formula. Now, we show that the bounded model construction procedure for M2L-STR can be used as an LTL bounded model checking procedure and furthermore, it produces a Boolean formula that is up to variable renaming syntactically equivalent to the formula resulting from applying the procedure of Biere et al.

Let M be a finite-state system, ϕ an LTL formula, and k a natural number. We designate with $[\![M, \phi, k]\!]$ the Boolean formula produced by the bounded model constructor of M2L-STR applied to the encoding $[\![M, \phi]\!]$ and k.

For d and k natural numbers we define the mapping $[\![.]\!]_k^d$ by

$$[\![M]\!]_k^d = I(s_0) \ \wedge \bigwedge_{0 \leq i < k-1} T(s_i, s_{i+1}) \ \wedge \ T(s_{k-1}, s_d).$$

The Boolean formula $[\![M]\!]_k^d$ is obtained from $\lceil M \rceil_l$ (see Section 5.3.4) by instantiating \$ with k and instantiating the free variable l by the natural number d.

For the natural numbers i, l and k we inductively define the mapping $[\![.]\!]_i^{l,k}$ by:

$$\begin{array}{rcl}
[\![p]\!]_i^{l,k} & = & L_p(s_i) \\
[\![\phi_1 \vee \phi_2]\!]_i^{l,k} & = & [\![\phi_1]\!]_i^{l,k} \vee [\![\phi_2]\!]_i^{l,k} \\
[\![\neg\phi]\!]_i^{l,k} & = & \neg[\![\phi]\!]_i^{l,k} \\
[\![\mathsf{next}\,\phi]\!]_i^{l,k} & = & \mathsf{if}\ i < k-1\ \mathsf{then}\ [\![\phi]\!]_{i+1}^{l,k}\ \mathsf{else}\ [\![\phi]\!]_l^{l,k} \\
[\![\phi_1 \mathsf{U} \phi_2]\!]_i^{l,k} & = & \bigvee_{j=i}^{k}([\![\phi_2]\!]_j^{l,k} \ \wedge \ \bigwedge_{x=i}^{j-1}[\![\phi_1]\!]_x^{l,k}) \ \vee \\
 & & \bigvee_{j=l}^{i-1}([\![\phi_2]\!]_j^{l,k} \wedge \bigwedge_{x=l}^{j-1}[\![\phi_1]\!]_x^{l,k} \wedge \bigwedge_{x=i}^{k}[\![\phi_1]\!]_x^{l,k})
\end{array}$$

The Boolean formula $[\![\phi]\!]_i^{l,k}$ is obtained from $\lceil \phi \rceil_i^{l,k}$ (see Section 5.3.4) by instantiating the variable i, l, and k by the natural numbers i, l, and k, by recursively applying the mapping ($[\![.]\!]_\cdot^{\cdot,\cdot}$) to the subformulae of ϕ, and applying standard Boolean simplification. We can derive the following rules for the time operators $\Diamond$ and $\Box$.

$$\begin{array}{rcl}
[\![\Diamond\phi]\!]_i^{l,k} & = & \bigvee_{j=\mathsf{min}(i,l)}^{k}[\![\phi]\!]_j^{l,k} \\
[\![\Box\phi]\!]_i^{l,k} & = & \bigwedge_{j=\mathsf{min}(i,l)}^{k}[\![\phi]\!]_j^{l,k}
\end{array}$$

We can prove inductively over the structure of the formula ϕ that the following theorem holds.

Theorem 5.4.2 $[\![M,\phi,k]\!] = \bigwedge_{0\leq l\leq k} [\![M]\!]^l_k \to [\![\phi]\!]^{l,k}_0$

Example 5.4.1 (continued)

$$\begin{aligned}
[\![M]\!]^0_3 &= (\overline{s^0_0} \wedge \overline{s^0_1} \ \wedge\ \overline{s^1_0} \wedge s^1_1 \ \wedge\ s^2_0 \wedge \overline{s^2_1}) \vee (\overline{s^0_0} \wedge s^0_1 \ \wedge\ s^1_0 \wedge \overline{s^1_1} \ \wedge\ s^2_0 \wedge s^2_1) \\
[\![\Diamond p]\!]^{0,3}_0 &= (\overline{s^0_0} \vee s^0_1) \vee (\overline{s^1_0} \vee s^1_1) \vee (\overline{s^2_0} \vee s^2_1) \\
[\![\Diamond r]\!]^{0,3}_0 &= (\overline{s^0_0} \wedge \overline{s^0_1}) \vee (\overline{s^1_0} \wedge \overline{s^1_1}) \vee (\overline{s^2_0} \wedge \overline{s^2_1})
\end{aligned}$$

The Boolean formula $[\![M]\!]^0_3 \to [\![\Diamond p]\!]^{0,3}_0$ is valid, whereas the Boolean formula $[\![M]\!]^0_3 \to [\![\Diamond r]\!]^{0,3}_0$ has the following counter-example $\{s^0_0 = 0, s^0_1 = 1, s^1_0 = 1, s^1_1 = 0, s^2_0 = 1, s^2_1 = 1\}$, which encodes the path $s_1 s_2 s_3$ of M. □

The formula $[\![M,\phi,k]\!]$ is, up to variable renaming, the result of applying the bounded model checker of Biere et al. to the problem instance (M,ϕ,k).

5.5 Chapter Summary

We investigated how we can reason about non-terminating systems in M2L-Str. We gave an embedding of LTL model checking in M2L-Str demonstrating:

- We can use finite automata on finite words instead of Büchi automata to decide the LTL model checking.
- In M2L-Str we can express safety as well as liveness properties.
- The bounded model constructor can be used as a bounded model checker without loss of efficiency.

In the practical side, the positive experimental results of the encoding of LTL satisfiability in Mona reported in [HH01] give us hope that similar positive results for our encoding of LTL model checking into Mona can be obtained. In the theoretical side, we still have to provide an analysis of the computational complexity. For the encoding of finite-state systems, we can prove that the size of the obtained finite automata is polynomial in the size of the finite-state systems provided as input. For the encoding of LTL formulae in M2L-Str, the picture is rather different. Here, we have to consider the syntactical fragment of M2L-Str, to which LTL is mapped, and explore the size of the finite automata that can be constructed from formulae of that fragment. This is, however, not so obvious.

Chapter 6

MonaCo

In this chapter we describe MO*NAC*O, *an implementation of the bounded model construction approach for* M2L-STR *and we report on numerous applications from diverse domains. Our aim is to evaluate the practical use and the scalability of* MO*NAC*O.

6.1 Introduction

In Chapter 4 we have investigated the bounded model construction problem for M2L-STR from the theoretical point of view. We proved that, given a M2L-STR formula ϕ and a natural number k, we can generate a formula in QBL that is satisfiable if and only if ϕ has a word model of length k. The generated formula is polynomial in the size of ϕ and k and can be tested for satisfiability in polynomial space. For generating length k counter-models, this yields a non-elementary improvement over the automata-based decision procedure for M2L-STR.

The aim of this chapter is to evaluate the bounded model construction for the logic M2L-STR in practice. To achieve this goal, we have implemented the tool MONACO, which supports the use of multiple quantified Boolean satisfiability solvers, *QSAT-solvers*, as a backend. We evaluated our system on several examples coming from different domains. Our experiments will show that MONACO compared to MONA usually finds counter-examples substantially faster.

The MONA system has two sources to abort the automata construction. The first source consists in the state-space explosion problem. Often, MONA fails to build the automaton for the corresponding formula and therefore it is not able to find errors, although, in most cases errors have small path lengths and require only a few number of states to be explored. Moreover, MONA uses a canonical form (deterministic and minimal automaton) to represent formulae. This design feature sometimes appears as an obstacle for successful verification with MONA, because deterministic automata can be exponentially larger than nondeterministic ones and thus, the construction of a deterministic automaton may fail, whereas the construction of its corresponding nondeterministic one may be possible. MONACO avoids the state-space explosion problem by only considering a fixed number of states. In MONACO, we can imagine the situation as follows. We can think about the formula as a nondeterministic automaton. First, MONACO removes all the states in the automaton that are reachable from the initial states in more than k steps. Second, it explores the resulting automaton, which is considerably smaller than the original one, for counter-examples.

The second source of divergence in MONA consists in the use of BDDs for representing the transition function of automata. BDDs often allow a compact representation of Boolean functions, but they strongly depend on the variable ordering. Consequently, MONA fails in many cases to build the transition function for automata of relatively small size. MONACO uses QSAT-solvers, which do not suffer from variable ordering and are capable to handle very large numbers of variables. Some of these solvers come with advanced search heuristics and have very good performance in practice as we will see in our experiments later.

The rest of this chapter is structured as follows. In Section 6.2 we describe MONACO system. In Section 6.3 we report on several applications. In Section 6.4 we draw conclusion.

The MONACO system is implemented in C++ and supports the use of different QSAT-solvers for QBL. The overall structure of the MONACO system is depicted in Figure 6.1. As input, it takes a file that contains the M2L-STR formula to be checked, a natural number as a model length (*bound*), a flag indicating whether we are looking for examples or counter-examples, and a QSAT-solver. The input file, also called (M2L-STR) *program*, is written in the MONA input syntax (*cf.* [KM01]). It consists of a list of declarations of variables and predicates followed by a formula, called *main formula* (see Example 6.2.1).

MONACO translates the main formula of the input M2L-STR program into a quantified Boolean formula that is subsequently passed to the specified QSAT-solver to be decided. The computation steps of MONACO can be grouped into five phases.

(1) This phase consists of parsing the input program and producing from the main formula a new (M2L-STR) formula in which the predicate calls are unfolded using their definitions.

(2) In this phase, MONACO eliminates the first-order and second-order quantifiers from the formula obtained in the first phase. Let k be the bound. The elimination of the first-order quantifications is performed as already described in Section 4.2.1. A formula of the form $\exists x.\, \phi$ (respectively $\forall x.\, \phi$) is unfolded

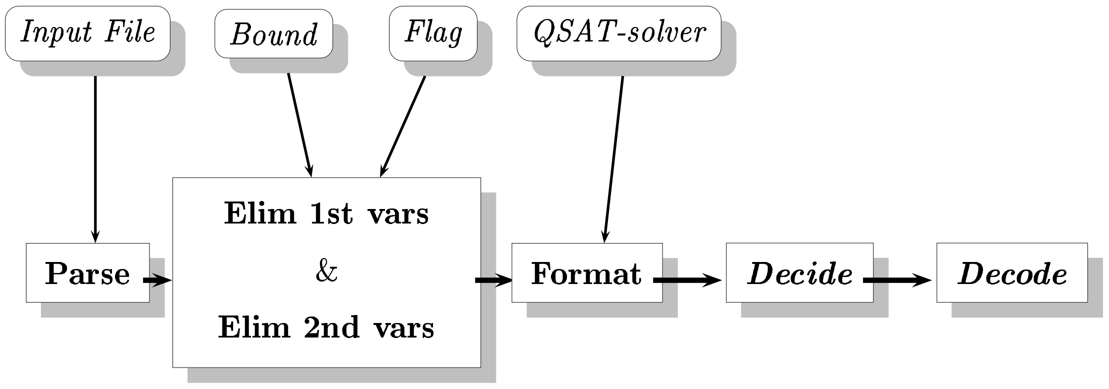


Figure 6.1: Structure of the MONACO System

into the formula $\bigvee_{0\leq i<k} \phi[i/x]$ (respectively $\bigwedge_{0\leq i<k} \phi[i/x]$). The elimination of the second-order variables introduces a block of Boolean quantifiers for every second-order variable. A formula of the form $\exists X.\, \phi$ (respectively $\forall X.\, \phi$) is translated into $\exists x_0, \ldots, x_{k-1}.\, \tilde{\phi}$ (respectively $\forall x_0, \ldots, x_{k-1}.\, \tilde{\phi}$), where $\tilde{\phi}$ is obtained by recursively eliminating the second-order variables from ϕ and replacing subformulae of the form $X(i)$, where i is a natural number, by the Boolean variable x_i if i is smaller than k and by false otherwise. This phase produces with a quantified Boolean formula.

(3) In this phase, MONACO translates the QBL formula obtained in the previous phase into the appropriate format. The format depends on the specified QSAT-solver. The system QUBOS (described in the next Chapter) for example works directly on fully-quantified Boolean formulae. Other QSAT-solvers, like SEMPROP [Let01] and QBF [Rin01], require the input format suggested by Rintanen in [Rin99b] which is an extension of the *DIMACS* format [JTI94].

(4) In this phase, MONACO invokes the QSAT-solver with the previously computed formula.

(5) In the last phase, MONACO decodes the assignment computed by the QSAT-solver into a model (respectively counter-model) of the input program.

6.2 Structural Description

Example 6.2.1 We consider a toy example to illustrate the usage of MONACO. The program under consideration is called `even.mona` and it is listed below.

```
m2l-str;
pred even(var1 p) =
  ex2 Q: p in Q
    & (all1 q:
        (0 < q & q <= p) =>
            (q in Q => q - 1 notin Q)
          & (q notin Q => q - 1 in Q))
    & 0 in Q;

# main Formula
var2 A;
all1 p : even(p) <=> p in A ;
```

The predicate `even` states that the natural number (first-order variable) `p` is even. The main formula states that the set `A` contains all the even natural numbers up to the implicit bound. We can call MONACO to obtain a set containing all the even numbers less than 10. This is done by invoking MONACO as follows

```
monaco even.mona 10 -s -qubos
```

and in this case MONACO responds with `A={0,2,4,6,8}`. The flag `-s` means that we are interested in a satisfying example and `-qubos` indicates that we are using QUBOS as QSAT-solver. We can also call MONACO to obtain a counter-example of length 10. We type

```
monaco even.mona 10 -c -qubos
```

responds with `A={1,3,5,7,9}`. The flag `-c` indicates that we are interested in a counter-example. Note that we can for example replace `-qubos` with `-semprop` (respectively `-qbf`) to use SEMPROP (respectively QBF) instead of the QUBOS. □

In the remainder of this chapter we will report on several applications that we have carried out using the MONACO system. In most of our examples we have used the MONA system for comparison. As we already mentioned in Chapter 1, MONA is an automata-based implementation of decision procedures for the monadic logics M2L-STR, WS1S and their generalizations to trees. MONA compiles a formula into a minimal deterministic automaton, which it represents and manipulates using BDDs. Over the last few years MONA has been continually improved and is now highly optimized (we use version 1.4).

For our tests, we used a 750 Mhz Sun Ultra Sparc workstation with 2 gigabytes RAM. The runtimes depicted in the tables below are *user time* (in seconds) reported by the operating system for all computations required. Times greater than one hour are indicated by the symbol abort. The space requirements depicted in the tables are in megabytes.

We used the QSAT-solvers QUBOS, SEMPROP, and QBF as backends for MONACO. Here, we will focus on comparing the bounded model checking approach versus the automata-based decision procedures for M2L-STR. A comparison of the impact of the use of these solvers is the subject of Chapter 7.

6.3 Applications

6.3.1 Small Examples

For completeness, we tested examples ranging from those that are easy for MONA to those that are difficult. Table 6.1 presents tests on several easy examples most of them distributed with MONA [KM01]. For all of these, we find counter-examples (of the same length as MONA's) when they exist. However, for these examples, MONA is much faster.

The first example is a parameterized *n-bit ripple-carry adder*, taken from [BK98]. The input formula states the equivalence between a structural description of the parameterized adder family (described at the gate level) with a behavioral description, describing how bit-strings are added. We checked this equivalence for $1 \leq k \leq 10$. The second example involves a structural specification of a sequential *D-type flip-flop* circuit, and its behavioral model. The circuit is built from 6 *nand*-gates, each of which has a (unit) time-delay. We tested the correctness of this circuit with respect to a behavioral description proposed by Gordon in [Gor86]. As has been discovered by [BK98, WP89], the specification has a subtle bug. Both MONA and our system find a (different) counter-example of length 8. The third example is a buggy version of the Dekker mutual exclusion protocol taken from [BA90]; both systems successfully find a trace showing that the critical sections can be simultaneously accessed.

Next we consider some examples that are difficult for MONA. First, we consider reasoning about two concurrent processes that increment a shared integer variable N by each executing the program: `Load Reg N`, `Add Reg 1`, `Store Reg N`. If we assume an interleaving semantics, it is possible that N is incremented by either 1 or 2. We model the two parallel processes in M2L-STR and assert (incorrectly) that after execution N is incremented by 1. Table 6.2 gives the results, where we scale the problem by considering registers of different bit-width. For more than 4 bits, MONA runs out of memory as the automata accepting the computations (traces) of the two systems grows exponentially.

Examples	Mona		MonaCo						
			k	Qubos		Qbf		Semprop	
	time	space		time	space	time	space	time	space
Invalid ripple-carry adder	0	0	1	0	0	0	0	0	0
			2	0	0	53	5	0	0
			3	0	0	abort		463	6
			4	0	0	abort		abort	
			5	0	0	abort		abort	
			6	2	5	abort		abort	
			7	18	10	abort		abort	
			8	48	15	abort		abort	
			9	385	83	abort		abort	
			10	1612	200	abort		abort	
Valid ripple-carry adder	0	0	1	0	0	0	0	0	0
			2	0	0	25	6	0	0
			3	0	0	abort		39	6
			4	0	0	abort		abort	
			5	0	0	abort		abort	
			6	1	5	abort		abort	
			7	3	7	abort		abort	
			8	13	13	abort		abort	
			9	55	28	abort		abort	
			10	286	71	abort		abort	
FlipFlop	0	0	8	0	0	0	0	0	0
Mutual exclusion	0	0	15	0	0	821	6	20	22

Table 6.1: Small Examples

$N \leq 2^i$	Mona		MonaCo $(k = 7)$					
			Qubos		QBF		Semprop	
	time	space	time	space	time	space	time	space
2	4	60	0	0	0	0	1	0
3	49	585	0	0	1	0	2	6
4	abort		0	0	3	0	34	7
5	abort		0	0	10	5	36	8
6	abort		1	0	34	8	108	11
7	abort		2	14	148	14	1219	23
8	abort		10	27	680	27	abort	
9	abort		39	55	abort		abort	
10	abort		197	118	abort		abort	
11	abort		1013	250	abort		abort	

Table 6.2: Parallel Instruction

width	Mona		MonaCo $(k = 8)$					
			Qubos		QBF		Semprop	
	time	space	time	space	time	space	time	space
2	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
8	0	0	0	0	20	15	3	7
16	abort		1	5	abort		146	9
32	abort		2	13	abort		abort	
64	abort		19	43	abort		abort	
128	abort		63	163	abort		abort	
256	abort		401	647	abort		abort	

Table 6.3: Barrel Shifter

width	Mona		MonaCo ($k = 15$)					
			Qubos		Qbf		Semprop	
	time	space	time	space	time	space	time	space
2	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
8	6	6	0	0	2	0	1	0
11	614	35	0	0	5	5	1	7
12	abort		0	0	7	5	1	5
16	abort		0	0	18	6	2	9
24	abort		1	5	73	9	6	13
32	abort		2	14	207	14	13	22
38	abort		5	18	387	19	21	28
40	abort		6	19	470	20	24	30
48	abort		11	26	932	27	41	40
50	abort		12	28	1082	29	47	43
54	abort		16	32	1496	34	63	49
60	abort		26	39	2612	40	86	59
64	abort		31	44	abort		107	63

Table 6.4: Counter

Finally, we consider two sequential circuits: a counter and a barrel shifter, which we parameterize in the width of the data-path. Tables 6.3 and 6.4 give the results of these experiments for data-paths of various widths. In the first example, the n-bit counter has two selection lines and n data lines. At each point in time the value of the data lines is incremented, reset, or unchanged depending on the value of the selection lines. We verify this with respect to an incorrect specification, which asserts that, after fifteen time units, the data line is always incremented. In our experiments, Mona quickly runs into state-space explosion problems, whereas even for large data-paths, MonaCo can still generate counter-examples quickly. Our procedure finds that, for data-paths between 4 and 64, the short counter-examples have length $k = 15$. The results for the barrel shifter are similar.

6.3.2 Alternating Bit Protocol

In this section we report on experiments with the alternating bit protocol (ABP) [BSW69] which has already received considerable attention in other approaches. Here, we first show how this protocol can be formalized using M2L-Str and second, we demonstrate how the MonaCo system can help to debug and establish

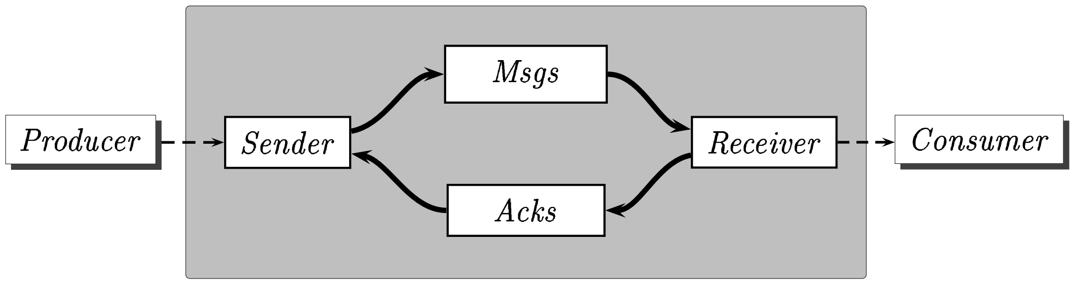


Figure 6.2: Alternating Bit Protocol

correctness of some safety properties of the protocol.

The ABP is designed to enable two entities, a producer and a consumer, to communicate in a reliable manner over an unreliable communication medium. The protocol constitutes of a sender, *Sender*, and a receiver, *Receiver*, which exchange messages over two logical channels *Msgs* and *Acks* which act as FIFO queues of unbounded capacity. Figure 6.2 displays the overall structure of the ABP. The sender reads the producer's messages and transmits them to the receiver which forwards them to the consumer. The sender sends the messages via the channel *Msgs* and waits for acknowledgments arriving via the channel *Acks*. If the sender does not receive the acknowledgment of a sent message then it resends the lost message so often until it receives an acknowledgment. To make two consecutive messages distinguishable, the sender marks each message with a bit. The receiver also marks each acknowledgment with a bit. When the sender receives an acknowledgment accompanied with a bit then it compares its own bit and the received bit. In the case that both bits do not coincide (which means that the previously sent data is lost) then it resends the data without changing the value of its own bit. In the case that both bits coincide (which means that the previously sent data is not lost) then the sender alternates its own bit and proceeds to send a new producer's data. Analogously, when the receiver receives a message accompanied with a bit that does not coincide with its own bit then it simply sends an acknowledgment accompanied with the negation of its own bit. In the other case, the receiver delivers the message to the consumer, updates its own bit, and sends an acknowledgment accompanied by the negation of its own bit.

Embedding the ABP in M2L-Str The alternating bit protocol represents an infinite-state system with the following two sources of infinity. First, the messages can be taken from an infinite data domain. Second, the used channels are of unbounded size. In [ACW90] it is shown that it is sufficient to consider only three different message values to establish the correctness of the ABP. It is also shown (*cf.*

[MN95]) that channels with a maximal size of two messages can be used instead of unbounded channels.

We have encoded ABP in M2L-Str and the complete protocol code is given in Appendix C.3. In our encoding, we model three different kinds of messages and the channels have size two. Here, we summarize the main ideas behind our model.

- The sender has a local variable, *SBit*, used to store the alternating bit, and three local states: *get*, *send*, and *wait*. The sender is initially in the state *get* where it gets data from the producer. Then it changes to the state *send_data* where it sends the producer's data, and finally moves to the state *wait* to wait for acknowledgment from the receiver and moves again to the state *get*.
- Analogously, the receiver has a local variable, *RBit*, and three local states: *receive, deliver*, and *send_ack*. The receiver is initially in the state *receive*, where it waits for data. When it reads data it moves to the state *deliver*, where it delivers data to the consumer. When the transmission of the data to the consumer is performed it moves to the state *send_ack* to send an acknowledgment and goes again to the state *receive*.
- The channel *Msgs* has an input and output interface. The data sent from the sender is collected at the input and is afterwards transfered to the output interface where it can be accessed by the receiver. Note that while transferring the data from the input to the output of the channel the data can be lost.
- Analogously, the channel *Acks* has also an input and output interface. The acknowledgments sent by the receiver are collected at the input and then transfered to the output where they are consumed from the sender. Note also that while transferring the acknowledgments from the input to the output of the channel the acknowledgments can also be lost.

Each of the processes *Sender, Receiver, Msgs, Acks* is defined by an M2L-Str predicate and the protocol which is a parallel composition of these four processes is specified by the disjunction

$$System \equiv Sender() \vee Receiver() \vee Msgs() \vee Acks()\,.$$

We have formalized and verified the following safety properties:

$$\begin{aligned}
\phi_1 \quad &\equiv \quad \forall s,t.\, s<t \wedge get(s) \wedge get(t) \wedge \forall q.\, s<q<t \rightarrow \neg get(q)\\
&\qquad \rightarrow \forall q.\, s<q<t \rightarrow SBit(s+1) \leftrightarrow SBit(q)\\
\phi_2 \quad &\equiv \quad \forall s,t.\, s<t \wedge \neg SBit(s) \wedge SBit(t) \wedge \forall q.\, s<q<t \rightarrow \neg SBit(q)\\
&\qquad \rightarrow \exists q.\, s\le q<t \rightarrow ack0(q)\\
\phi_3 \quad &\equiv \quad \forall s,t.\, s<t \wedge SBit(s) \wedge \neg SBit(t) \wedge \forall q.\, s<q<t \rightarrow SBit(q)\\
&\qquad \rightarrow \exists q.\, s\le q<t \rightarrow ack1(q)
\end{aligned}$$

Properties	Mona	MonaCo						
			Qubos		Qbf		Semprop	
	time	k	time	space	time	space	time	space
ϕ_1	abort	10	0	0	8	0	76	8
		20	1	0	83	10	abort	
		30	7	0	2734	78	abort	
		40	89	40	abort		abort	
		50	439	187	abort		abort	
		60	2249	398	abort		abort	
ϕ_2	abort	10	0	0	7	0	0	0
		20	0	0	48	20	6	0
		30	2	0	195	32	24	5
		40	4	10	629	51	81	10
		50	9	15	abort		abort	
		60	17	23	abort		abort	
		70	31	50	abort		abort	
		80	51	84	abort		abort	
		90	89	157	abort		abort	
		100	339	201	abort		abort	
		110	779	246	abort		abort	
		120	791	298	abort		abort	
ϕ_3	abort	10	0	0	7	0	0	0
		20	0	0	48	30	4	8
		30	2	8	192	54	20	17
		40	4	13	607	84	76	43
		50	9	56	abort		abort	
		60	16	74	abort		abort	
		70	48	93	abort		abort	
		80	184	105	abort		abort	
		90	708	163	abort		abort	
		100	1330	197	abort		abort	

Table 6.5: Experimental Results of the Alternating Bit Protocol

Formula ϕ_1 describes that the alternating bit of a sender does not change during the transition from a get state to the next following get state. The formulae ϕ_2 and ϕ_3 formalize that if the alternating bit of the sender changes once during a time interval between s and t (the bit changes from *false* to *true* in ϕ_2 and from *true* to *false* in ϕ_3) then the sender received an acknowledgment within that interval of time and this asserts that the consumer has indeed received the data.

The results of our experiments are depicted in Table 6.5. The MONA system was not able to build even the DFA corresponding to the protocol, and consequently to check the properties given above. Using MONACO we were able to debug our encoding and to remove a lot of errors of several length. The results displayed in Table 6.5 show the timings and space requirements taken by MONACO after removing the bugs.

6.3.3 The Bus Arbiter Protocol

In this section we use the *bus arbiter circuit* [Mar85, Dil88] to demonstrate the scalability of the MONACO system.

The bus arbiter is an arbitration protocol designed to grant access to only one client among a number, N, of clients that contend for the use of a bus. The clients send requests to the arbiter which responds with acknowledgments. To assert that every request is eventually acknowledged, the protocol uses a ring of cells, one cell per client[1], and circulates a token in the ring. A client is granted immediate access to the bus, when its request persists for the time it takes for the token to make a complete circuit.

A cell of the arbiter is depicted in Figure 6.3. The inputs *token_in*, *grant_in*, and *override_in* are directly connected to the outputs *token_out*, *grant_out*, and *override_out* respectively of the previous cell in the ring and the outputs *token_out*, *grant_out*, and *override_out* form the inputs of the next cell in the ring. The cell contains of two registers; register T to store the token and register W to store information if the cell is waiting for acknowledgment since the last pass of the token. The signal *grant_out* of a cell is set (high) exactly when all the previous cells in the ring are not requesting the bus. The signal *override_out* is used to prevent the other cells from being acknowledged at the same time. A cell is acknowledged when it is requesting the bus and no other cell is requesting the bus or it is the first one requesting the bus since one token loop.

Embedding the Bus Arbiter in M2L-Str The code of the bus arbiter protocol for three cells is given in Appendix C.4 The circuit cell is modeled in M2L-STR straightforwardly using the following predicate.

[1]We will use the word cell and client interchangeably.

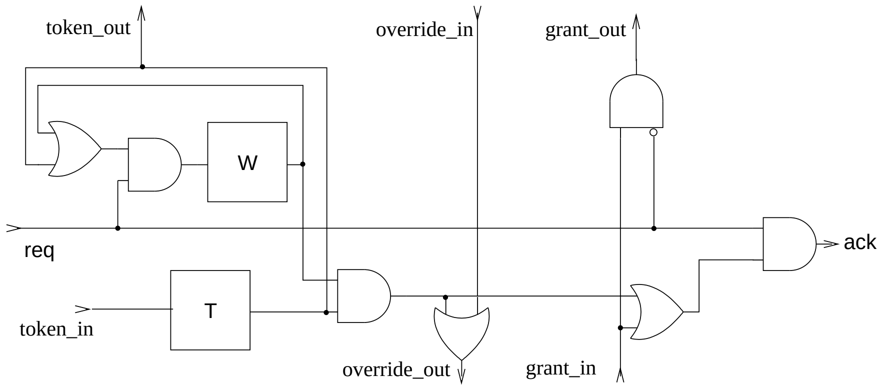


Figure 6.3: Bus Arbiter Cell

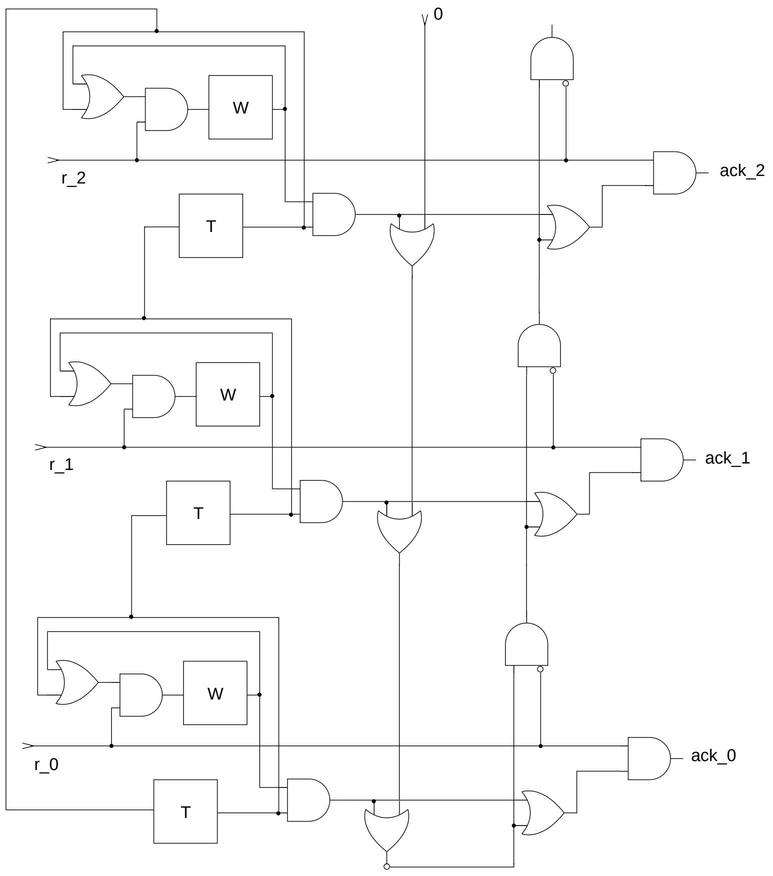


Figure 6.4: Bus Arbiter of 3 Celles

$$\begin{array}{l}
\textsf{pred}\ Cell(\textsf{var0}\ request, token, n\_token, wait, n\_wait, token\_in,\\
\qquad\qquad token\_out, grant\_in, grant\_out, override\_in, override\_out, ack) =\\
\quad \begin{array}{lcl}
(n\_token & \leftrightarrow & token\_in) \wedge\\
(n\_wait & \leftrightarrow & (wait \vee token) \wedge request) \wedge\\
(ack & \leftrightarrow & (wait \wedge token \vee grant\_in) \wedge request) \wedge\\
(token\_out & \leftrightarrow & token) \wedge\\
(grant\_out & \leftrightarrow & \neg\, request \wedge grant\_in) \wedge\\
(override\_out & \leftrightarrow & wait \wedge token \vee override\_in)
\end{array}
\end{array}$$

The Boolean variables *n_token* and *n_wait* are used to model the content of the registers T and W respectively at the next time unit.

The bus arbiter circuit for 3 clients is displayed in Figure 6.4. Initially, only one register T is set and all W registers are reset. We have implemented the bus arbiter for $N \leq 12$ clients. Below, we display the M2L-Str predicate formalizing the arbiter for 3 clients.

$$\begin{array}{l}
\textsf{pred}\ Arbiter(\textsf{var2}\ R_0, R_1, R_2, A_0, A_1, A_2, T_0, T_1, T_2, W_0, W_1, W_2,\\
\qquad\qquad Tin_0, Tin_1, Tin_2, Tout_0, Tout_1, Tout_2, Oin_0, Oin_1, Oin_2,\\
\qquad\qquad Oout_0, Oout_1, Oout_2, Gin_0, Gin_1, Gin_2, Gout_0, Gout_1, Gout_2) =\\
\quad Init\_cell0(T_0, W_0) \wedge\\
\quad Init\_cell(T_1, W_1) \wedge\\
\quad Init\_cell(T_2, W_2) \wedge\\
\quad \forall\, t.\, First\_last\_wiring(t, \ldots) \wedge\\
\qquad Cell(t \in R_0, \ldots) \wedge\\
\qquad Cell(t \in R_1, \ldots) \wedge\\
\qquad Cell(t \in R_2, \ldots)
\end{array}$$

The second-order variables R_0, …, $Gout_2$ model the input and output signals of the cells. The predicate *Init_cell0* states the initial values of the registers of the first cell. The two other predicates *Init_cell* state the initial values of the registers of the second and third cell. The predicate *First_last_wiring* describes how the inputs of the first cell and the outputs of the last cell are connected.

We have formalized the following three properties in M2L-Str:

1. At each time point the token is exactly at one cell.

2. Only a single client is allowed to use the bus.

3. There is no assertion without a request.

#clients	MONA		MONACO						
			k	QUBOS		QBF		SEMPROP	
	time	space		time	space	time	space	time	space
2	0	0	12	0	0	0	0	0	0
3	3	62	28	2	11	125	12	9	11
4	84	1080	68	188	157	abort		abort	
5	abort		50	63	77	abort		abort	
	abort		75	257	556	abort		abort	
	abort		100	2682	604	abort		abort	
6	abort		25	5	14	abort		abort	
	abort		50	128	92	abort		abort	
	abort		75	1040	250	abort		abort	
7	abort		25	7	17	abort		abort	
	abort		50	193	104	abort		abort	
	abort		75	1582	343	abort		abort	
8	abort		25	8	19	abort		abort	
	abort		50	277	119	abort		abort	
	abort		75	2198	385	abort		abort	
9	abort		25	13	21	abort		abort	
	abort		50	341	133	abort		abort	
	abort		75	2786	440	abort		abort	
10	abort		25	18	23	abort		abort	
	abort		50	415	148	abort		abort	
	abort		60	1154	249	abort		abort	
11	abort		25	20	25	abort		abort	
	abort		50	590	159	abort		abort	
	abort		60	1503	272	abort		abort	
12	abort		25	21	27	abort		abort	
	abort		50	755	175	abort		abort	
	abort		60	1928	299	abort		abort	

Table 6.6: Experiment Results for the Bus Arbiter

```
class Test {                     Method int fac(int)
                                >> max_stack=4, max_locals=2 <<
int fac (int i) {                   0 iload_1
    if ( i == 0 )                   1 ifne 4
      return 1;                     2 iconst_1
    else                            3 ireturn
      return i*fac(i-1);            4 iload_1
  }                                 5 aload_0
}                                   6 iload_1
                                    7 iconst_1
                                    8 isub
                                    9 invokevirtual <Test.fac(int):int>
                                   10 imul
                                   11 ireturn
```

Figure 6.5: Java code and bytecode of method fac

We have used MONA and MONACO to determine whether the arbiter circuits for up to 12 clients have the above properties. The results are depicted in Table 6.6. Neither MONA nor MONACO detected errors for all these problems. For $N \leq 4$, the time requirements for both systems are slightly different, however, the MONA system needs considerably more space than MONACO. The chosen bounds for the three first circuits coincide with the number of the states of the minimal DFA of the corresponding circuit. Note that in this case we can also deduce from the results of MONACO that the three circuits are correct with respect to the specified properties. That is, no errors can be detected for any bound $k \in \mathbb{N}$. By increasing the number of clients MONA quickly runs out of memory.

6.3.4 Bytecode Verification

In this section we will demonstrate the usability of MONACO in a very interesting domain, namely bytecode verification. We briefly overview the bytecode verification approach and describe a new backend for MONACO to the bytecode verifier BYCOMOCHE developed by Basin et al. [BFGP02]. We will show that the new backend somehow "completes" BYCOMOCHE and report on experimental results. We refer to [Ler01] for an overview on bytecode verification.

Given a Java source file, the Java compiler produces a class file containing *bytecode* (see Figure 6.5), which is executed thereafter by the Java virtual machine (JVM). In today's mobile applications, it is widespread that untrusted bytecode is downloaded and locally executed which introduces major security risks such as destroying and modifying sensible data or even distributing private information in the network. Bytecode verification is the task of statically checking that the bytecode satisfies certain safety properties that ensure that no bad things can happen prior to executing the code.

The Java virtual machine is a stack-based abstract machine. It has a stack to store the operands of instructions and registers to hold intermediate results as well as local variables. One safety property that bytecode should meet is *type correctness*: When the JVM executes a bytecode instruction, it checks that the operands of that instruction stored at the top of the stack are of the appropriate types. For example, when the JVM executes the instruction `isub` (integer subtraction, see Figure 6.5), it should check that the two top elements of the stack are of type `int`. Another safety property is that there is no stack-overflow when a bytecode instruction is executed.

As mentioned before, the widespread use of Java and the increased demand on secure Java have motivated a large number of researchers to work on bytecode verification. There is a considerable number of approaches proposed for bytecode verification. Almost all these approaches are based on a type-level abstract interpretation of the JVM. By this abstract interpretation, a bytecode instruction does not operate on values but on types. For instance, the abstract interpretation of the instruction `isub` is an operation which applied to the pair (`int`, `int`) produces the type `int`. So, in the abstract JVM, the stack and registers store types instead of values. Verifying the type correctness property for the instruction `isub` results in checking that by executing this instruction the two top elements of the stack are type `int` and that the type of the top element of stack after the execution is also `int`. Following the abstract interpretation, a bytecode method can be translated into a finite-state transition system where the contents of the stack and the registers build the finite state-space and the abstract instructions define the transition function.

New Backend to ByCOMOCHE and Experimental Results

There exist diverse approaches to bytecode verification [Sch98, Coh96, NvO98], here we concentrate on BYCOMOCHE. The BYCOMOCHE tool uses an intermediate representation of the abstract transition system and comes with a backend for SPIN and a backend for SMV. In the SPIN backend, the transition system is described as a PROMELA process and the properties are expressed in the linear temporal logic (LTL) as an observer process. In the SMV backend, the transition system is described as a process in the input language of SMV and the safety properties are expressed in the computational temporal logic (CTL). The experimental results performed using BYCOMOCHE and reported in [BFGP02] showed that bytecode verification based on model checking is as competitive as bytecode verification based on data-flow analysis for correct bytecode. The reason for this positive result, although model checkers have in the worst case exponential time complexity, is due to the fact that for correct bytecode only linearly many states in the number of instructions are reachable. Moreover, the experiments in [BFGP02] showed that in the case of correct bytecode SPIN has a better performance than SMV that demonstrates that explicit model checking using the "on-the-fly" technique is more appropriate for this

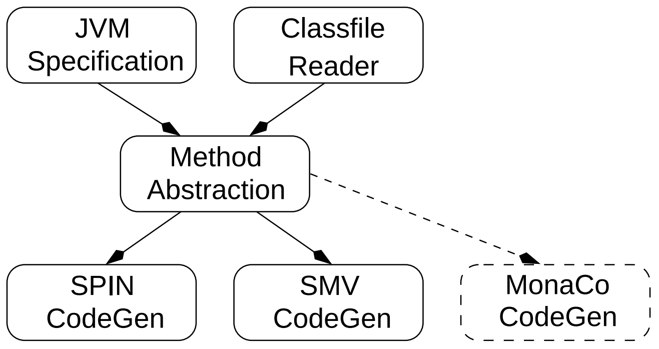


Figure 6.6: Extension of ByCoMoChe

kind of task than symbolic BDD-based model checking.

For incorrect bytecode the picture is rather different. Both Spin and Smv are of limited success. Spin often fails to terminate and Smv runs out of memory. The analysis of this problem given in [BFGP02] indicates that for incorrect bytecode the number of reachable states could be exponentially larger than in the case of correct bytecode which makes the verification task very resource intensive and thus aggravates detecting errors, even with small paths. For detecting errors, we alternatively use the approach of bounded model construction. We have added a new backend for MonaCo to the bytecode verifier ByCoMoChe. The overall structure of the new obtained system is depicted in Figure 6.6. The backend for MonaCo generates from the abstract transition system an M2L-Str formula in two steps. First, it produces a linear Lisa code (*cf.* Section 3.6). Second, it uses the Lisa compiler to generate an M2L-Str formula that is subsequently processed by MonaCo. In Lisa, the types of the abstract transition system are modeled as integers, the stack and registers are modeled as arrays of fixed length. In Figure 6.5 we have displayed the bytecode of the function *fac* and in Figure 6.7 the generated Lisa code.

We carried out two different kinds of experiments. First, we have applied the Mona system to a large number of examples of both correct and incorrect bytecode. The results confirm the thesis claimed above, namely the “on-the-fly” technique is better than the symbolic BDD-based model checking in bytecode verification. In our experiments, Mona, which uses BDDs in its implementation, was able to only verify a few examples with small numbers (up to 10) of instructions.

Second, we performed experiments with bytecode and compared the three backends. The results are displayed in Figure 6.8 and show that for faulty bytecode the use of the MonaCo backend is the best choice. Note that the bytecode methods used in these experiment are originally correct, we produced defective bytecode.

```
# type declarations

data typesT = 0..3;
data pcT = 0..11;
data locindexT = 0..1;
data locT = array locindexT  of  typesT;
data stack_indexT = 0..4;
data stackT = array stack_indexT  of  typesT;
data stateT = record {pc:pcT, stack_ptr:stack_indexT, stack_:stack_T, loc:locT};
data runT = array nat  of  stateT;

# transition system
pred Init(S:stateT) =  S.loc[0] = 3 & S.loc[1] = 1 & S.stack_ptr = 0 & S.pc = 0;
pred Trans(S, S':stateT) =
  case (S.pc) of
  0 => S' = S{stack[S.stack_ptr] := S.loc[1], stack_ptr := S.stack_ptr+ 1,
            pc := 1};
  1 => (S' = S{stack_ptr := S.stack_ptr-1, pc := 2}) |
       (S' = S{stack_ptr := S.stack_ptr-1, pc := 4});
  2 => S' = S{stack[S.stack_ptr] := 1, stack__ptr := S.stack_ptr + 1,
            pc := 3};
  3 => S' = S;
  4 => S' = S{stack[S.stack_ptr] := S.loc[1], stack_ptr :=(S.stack_ptr) + 1,
            pc := 5};
  5 => S' = S{stack[S.stack_ptr] := S.loc[0], stack_ptr :=(S.stack_ptr) + 1,
            pc := 6};
  6 => S' = S{stack[S.stack_ptr] := S.loc[1], stack_ptr :=(S.stack_ptr) + 1,
            pc := 7};
  7 => S' = S{stack[S.stack_ptr] := 1, stack__ptr := S.stack_ptr + 1,
            pc := 8};
  8 => S' = S{stack[S.stack_ptr-2] := 1, stack__ptr := S.stack_ptr-1,
            pc := 9};
  9 => S' = S{stack[S.stack_ptr-2] := 1, stack__ptr := S.stack_ptr-1,
            pc := 10};
  10 => S' = S{stack[S.stack_ptr-2] := 1, stack__ptr := S.stack_ptr-1,
             pc := 11};
  11 => S' = S;
  esac;

# properties
pred Invariants(S:stateT) =
  case (S.pc) of
  0 => S.loc[1] = 1;
  1 => S.stack[S.stack_ptr-1] = 1;
  2 => true;
  3 => S.stack[(S.stack_ptr)-1] = 1;
  4 => S.loc[1] = 1;
  5 => S.loc[0] = 2 | S.loc[0] = 3;
  6 => S.loc[1] = 1;
  7 => true;
  8 => S.stack[S.stack_ptr-2] = 1) & S.stack[S.stack_ptr-1] = 1);
  9 => S.stack[S.stack_ptr-1] = 1) & S.stack[S.stack_ptr-2] = 3);
  10 =>S.stack[S.stack_ptr-2] = 1) & S.stack[S.stack_ptr-1] = 1);
  11 =>S.stack[S.stack_ptr-1] = 1;
  esac;

#main Formula

var runT R;
Init(R[0]) & all nat i: (0 < i) -> Trans(R[i-1], R[i]) ->
all nat i: Invariants(R[i]);
```

Figure 6.7: Abstract transition system for the fac bytecode in Lisa

Methods	Spin	Smv	MonaCo	
	time	time	time	k
Character_S_clinit___V	abort	abort	44	19
FDBigInt_longValue__J	abort	5	65	40
FDBigInt_lshiftMe_I_V	abort	abort	267	15
FDBigInt_multaddMe_II_V	abort	abort	71	21
FDBigInt_S_init__J[CII_V	abort	abort	15	40
FDBigInt_mult_I_Ljava_lang_FDBigInt	abort	abort	30	9
FDBigInt_quoRemIteration_Ljava_lang_FDBigInt_I	abort	abort	576	30
FDBigInt_add_Ljava_lang_FDBigInt_Ljava_lang_FDBigInt	abort	abort	47	15
FDBigInt_mult_Ljava_lang_FDBigInt_Ljava_lang_FDBigInt	abort	abort	93	20
FDBigInt_sub_Ljava_lang_FDBigInt_Ljava_lang_FDBigInt	abort	abort	179	40
StringBuffer_append_C_Ljava_lang_StringBuffer	abort	abort	1034	50
StringBuffer_insert_I[C_Ljava_lang_StringBuffer	abort	45	564	37
StringBuffer_insert_I[CII_Ljava_lang_StringBuffer	abort	12	1618	50
String_getBytes_Lsun_io_CharToByteConverter_[B	abort	abort	1368	50
String_regionMatches_ILjava_lang_StringII_Z	abort	abort	3694	50
String_S_init__[BIILsun_io_ByteToCharConverter_V	abort	23	98	17
String_toLowerCase_Ljava_util_Locale_Ljava_lang_String	0	abort	134	50
String_toUpperCase_Ljava_util_Locale_Ljava_lang_String	0	abort	344	50
Short_decode_Ljava_lang_String_Ljava_lang_Short	abort	33	53	10
Integer_toString_II_Ljava_lang_String	abort	0	7	5
Integer_toString_I_Ljava_lang_String	abort	abort	70	35
Integer_S_clinit___V	abort	abort	86	18
Integer_decode_Ljava_lang_String_Ljava_lang_Integer	abort	abort	812	20
Integer_parseInt_Ljava_lang_StringI_I	abort	abort	83	35
Long_decode_Ljava_lang_String_Ljava_lang_Long	abort	abort	62	23
Long_toString_JI_Ljava_lang_String	abort	32	10	5
Long_parseLong_Ljava_lang_StringI_J	abort	abort	118	10
Object_toString__Ljava_lang_String	abort	0	2	10
Object_wait_JI_V	0	abort	50	11

Figure 6.8: Bytecode Verification (buggy code)

6.4 Chapter Summary

Our exhibition of the experimental results in this chapter give evidence on the practicality of the bounded model construction for M2L-Str. We demonstrated for a large suite of examples, taken form divers domains, that MonaCo, the system implementing M2L-Str bounded model checking, provides a more efficient alternative to counter-example generation than using standard automata-theoretic decision procedures like Mona.

The application of MonaCo to bytecode verification successfully shows that the bounded model construction for M2L-Str can be used in a complementary way to other approaches like the LTL on-the-fly model checking, implemented in Spin, and like the symbolic model checking for CTL, implementend in Smv. Using the MonaCo system we were able to discover errors in buggy bytecode which was not possible to achieve using Spin and Smv.

MonaCo supports the use of multiple QSAT-solvers. The choice of the solver to be used as backend has an enormous impact on the performance of MonaCo. A detailed comparison of the best QSAT-solvers is postponed to Chapter 7. Here, we restrict ourselves to mentioning that the capability of MonaCo to use more than one backend has the advantage that any optimization and improvement of a backend necessarily leads to an improvement of the performance of MonaCo.

Chapter 7

Qubos: Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers

We describe QUBOS *(QUantified BOolean Solver), a decision procedure for quantified Boolean logic. The procedure is based on nonclausal simplification techniques that reduce formulae to a propositional clausal form after which off-the-shelf satisfiability solvers can be employed. We show that there are domains exhibiting structure for which this procedure is very effective and we report on experimental results.*

7.1 Introduction

In recent years there has been considerable work on developing and applying satisfiability (SAT) solvers for quantified Boolean logic (QBL). Applications include program verification using bounded model checking [BCCZ99] and bounded model construction (see Chapter 4), hardware applications including testing and equivalence checking [SB01], and artificial intelligence tasks like planning [Rin99a].

Solvers for (unquantified) Boolean logic have reached a state of maturity; there are many success stories where SAT-solvers such as [MMZ$^+$01, Sta89, Zha97] have been successfully applied to industrial scale problems. However, the picture for QBL is rather different. Despite the growing body of research on this topic, the current generation of Q(uantified)SAT-solvers [GNT01b, Let01, Rin99b] are still in their infancy. These tools work by translating QBL formulae to formulae in a quantified clausal normalform and applying extensions of the Davis-Putnam method to the result. The extensions concern generalizing Davis-Putnam heuristics such as unit-propagation and backjumping. These tools have not yet achieved the successes that

SAT tools have and our understanding of which classes of formulae these procedures work well on, and why, is also poor.

In this thesis, we present a different approach to the QSAT problem. It arose from our work in bounded model construction for monadic second-order logics (see Chapter 4) where we reduce the problem of finding small models for monadic formulae to QBL satisfiability. Our experience with available QBL solvers was disappointing. Their application to formulae involving more than a couple quantifier iterations would often fail, even for fairly simple formulae. In particular, our model construction procedure generates formulae where the scope of quantification is generally small in proportion to the overall formula size and in many cases quantifiers can be eliminated, without blowing up the formulae, by combining quantifier elimination with simplification. This motivated our work on a procedure based on combining miniscoping (pushing quantifiers in, in contrast to out, which is used in clause based procedures), quantifier expansion, and eager simplification using a generalization of Boolean constraint propagation. The transformation process is carried out until the result has only one kind of quantifier remaining, at which point the result can be converted to clausal form and given to an off-the-shelf (Boolean) SAT-solver.

Our thesis in this chapter is that our decision procedure works well (it is superior to other state-of-the-art approaches) when certain kinds of structure are present in the problems to be solved. Our contribution is to identify a notion of structure based on relative quantifiers scope, to show that certain classes of problems will naturally have this structure (*i.e.*, that the ideas presented in this chapter have general applicability), and to validate our thesis experimentally. Our experimental comparison is on two sets of problems, those arising in bounded model construction, which always exhibit significant structure, and those arising in conditional planning, which have varying degrees of structure.

Organization. The rest of the chapter is organized as follows. In Section 7.2, we introduce notation. In Section 7.3, we explain what kind of structure we will exploit and why certain classes of problems are naturally structured. In Section 7.4, we introduce our procedure and in Section 7.5, we present experimental results. We report on related work in Section 7.6 and finally, we draw conclusions in Section 7.7.

7.2 Background

The logics BL and QBL are introduced in Section 2.3. There, we have described the syntax of their formulae and explained in which sense QBL is an extension of BL. Here, we introduce some additional notion.

As notational shorthand, we allow quantification over sets of variables and we write $Qx_1, \ldots, x_n.\, \phi$ for the formula $Qx_1. \cdots Qx_n.\, \phi$, where $Q \in \{\forall, \exists\}$. We denote by $\mathsf{free}(\phi)$ the set of free variables in ϕ. Unless indicated otherwise, by "formulae" we mean quantified Boolean formulae instead of (unquantified) Boolean formulae.

A formula x or $\neg x$, where x is a variable, is called a *literal*. A formula ϕ is in *negation normalform* (*nnf*), if, besides the quantifiers, it contains only the connectives $\vee$, $\wedge$ and $\neg$, and $\neg$ appears only before variables. A formula ϕ is in *prenex normalform* (*pnf*) if it has the form $Q_1 X_1 \cdots Q_k X_k.\, \psi$ where $Q_i \in \{\exists, \forall\}$, each X_i is a finite set of variables, and ψ is a Boolean formula called the *matrix* of ϕ. A formula ϕ is in *quantified clausal normal form* (*qcnf*) if it is in *pnf* and its matrix is a conjunction of disjunctions of literals. We define the *prefix-type* of a formula in *pnf* inductively as follows. A Boolean formula has the prefix-type $\Sigma_0 = \Pi_0$. A formula $\forall x.\, \phi$ has the prefix-type Π_{n+1} (respectively Π_n) if ϕ has the prefix-type Σ_n (respectively Π_n). A formula $\exists x.\, \phi$ has the prefix-type Σ_{n+1} (respectively Σ_n) if ϕ has the prefix-type Π_n (respectively Σ_n). Finally, the *size* of a formula ϕ, denoted by $|\,\phi\,|$, is the number of variable occurrences, connectives and (maximal) quantifier blocks in ϕ, *i.e.*, the size of the abstract syntax tree for ϕ, where like quantifiers are grouped in blocks and only counted once.

7.3 Structured Problems

Our thesis is that our decision procedure works well (in particular, it is superior to other state-of-the-art approaches) when certain kinds of structure are present in the problems to be solved. In this section we explain what structure is, how one measures it, and why certain classes of problems will naturally have this structure.

The structure we exploit is based on a notion of quantifier scope, in particular the size of quantified subterms relative to the size of the entire term. When the average quantifier scope is small, our transformations can often successfully eliminate quantifiers in manageable time and space.

In our experiments, it is important to be able to measure structure to assess its effects on the decision procedure's performance. Our measure is based on the average quantifier weight, defined as follows:

Definition 7.3.1 *Let ϕ be a quantified Boolean formula, $Q \in \{\forall, \exists\}$, M_Q be the multiset of all Q-quantified subformulae of ϕ, and $\psi \in M_Q$. The* relative Q-weight of ψ with respect to ϕ *is*

$$\mathsf{rw}_Q^{\phi}(\psi) = \frac{|\,\psi\,|}{|\,\phi\,|}.$$

The average Q-weight of ϕ *is* $\mathsf{aw}_Q(\phi) = \frac{1}{|M_Q|}\Sigma_{\psi \in M_Q} \mathsf{rw}_Q^{\phi}(\psi)$.

Now, well-structured formulae are those with either a small average $\forall$-weight or small average $\exists$-weight (typically under 5%, as we will see for the first problem domain we consider), *i.e.*, those in which, for at least one of the quantifiers, quantified variables have small scopes on average. In contrast, poorly structured formulae with large average weight have many quantifiers with large scopes.

The two domains we consider are system verification using bounded model construction [AB00], and conditional planning [Rin99a]. For the first domain, we show that problems are *always* well-structured. In the second domain, the degree of structure varies considerably. The corresponding effectiveness of our decision procedure also varies in relationship to this structure.

7.3.1 Bounded Model Construction

Bounded model construction (BMC), as we have seen in Chapter 4, is a method for generating models for a monadic formula by reducing its satisfiability problem to a QBL satisfiability problem. Here, we will present a small example to show how structured problems arise in BMC and then explain why this is generally the case. We reconsider Example 2.7.1 where we specified and verified a parameterized family of ripple-carry adder. We recall that the equivalence between the specification and the implementation of the adder is stated by the formula

$$\begin{aligned}\Phi \equiv \forall n.\, \forall A, B, S.\, \forall cin, cout.\, \mathsf{adder}(n, A, B, S, cin, cout) \leftrightarrow \\ \mathsf{spec}(n, A, B, S, cin, out)\,.\end{aligned}$$

In this example, MONACO, the implementation of BMC, takes as input $\neg\Phi$ and a natural number k and produces a quantified Boolean formula as we described in Section 6.1 (phase 2). In this transformation, MONACO essentially unfolds the first-order quantifications k-times and replace the second-order quantifications with Boolean quantifications. This kind of transformation produces a quantified Boolean formula whose size is $O(k^2 \mid \phi \mid)$ in the bound k and original formula ϕ (see Theorem 4.2.6). In general, applications to practical verification problems give rise to large quantified Boolean formulae often on the order of 20 megabytes for larger examples, that we have tackled. Central to our approach here is the fact that the transformation *always* produces formula with a large amount of structure, as we explain below.

In the above transformation, large formulae (due to the k^2 factor in the expansion) result from expanding first-order quantification. In this example, we quantify outermost over n in stating our correctness theorem and this is always the case when verifying theorems about parameterized systems. Similarly, when reasoning about time dependent systems, like sequential circuits or protocols, one also always quantifies outermost over n, which represents time or the number of steps. The unfolding of this outermost quantifier alone explains the main reason why MonaCo results in a quantified Boolean formula of small average quantifier weight since, after the unfolding, the remaining quantified subformulae have a relative weight at most $1/k$ of the original formula. The unfolding of additional first-order quantifiers only serves to further reduce the average weight. Hence we have:

Lemma 7.3.2 *Let $\Phi \equiv \mathcal{Q}n.\,\phi$ be a first-order quantified monadic formula where $\mathcal{Q} \in \{\forall, \exists\}$ and let Φ' (respectively ϕ') be the result of the* MonaCo *expansion of Φ (respectively ϕ) with bound $k \in \mathbb{N}$. It holds that $\mathsf{aw}_Q(\Phi') = \frac{1}{k}\mathsf{aw}_Q(\phi')$, for $Q \in \{\forall, \exists\}$.*

Of course, MonaCo also eliminates second-order quantification, where a second-order quantifier is replaced with a block of Boolean quantifiers. In general, this has a negligible effect on the amount of structure since, after the outermost unfolding, these quantifiers have small relative scope. It follows then that MonaCo produces well-structured problems. Moreover, there is a positive correlation between problem size (resulting from large values of k) and structure, which helps to explain the good performance of our decision procedure on problems in this class.

7.3.2 Conditional Planning in Artificial Intelligence

The second problem domain that we use for experiments is conditional planning in QBL. A conditional planning problem is the task of finding a finite sequence of actions (which comprise a plan) whose successive application, starting from an initial state, leads to a goal state. Applications of conditional planning include robotics, scheduling, and building controllers. The main difference between conditional and classical planning is that the initial states as well as the moves from one state to another state depend on different circumstances that can be tested. This leads to interesting QBL problems. As shown in [Rin99a], finding a solution for a conditional planning problem can be expressed as a satisfiability problem for a quantified Boolean formula of the form:

$$\mathcal{P} \equiv \exists P_1, \ldots, P_m.\, \forall C_1, \ldots, C_n.\, \exists O_1, \ldots, O_p.\, \Phi\ .$$

The validity of the formula $\mathcal{P}$ means that there is a plan (represented by the variables $P_1, \ldots, P_m$) such that for any contingencies (represented by the variables $C_1, \ldots, C_n$) that could arise, there is a finite sequence of operations $(O_1, \ldots, O_p)$ whose applications allow one to reach the goal state starting from an initial state. The body Φ is a conjunct of formulae stating the initial states, goal states, and the next-state relation.

If $n = 0$ then $\mathcal{P}$ encodes a classical (non-conditional) planning problem. In this case, the validity of $\mathcal{P}$ can be checked using a SAT-solver. In the $n \neq 0$

```
proc QUBOS($\phi$, $SAT$) ≡
  let $Q \in \{\forall, \exists\}$ be the quantifier kind with smallest $\mathsf{aw}_Q$
  while ($\phi$ contains $Q$'s) do
    miniscope the quantifiers in $\phi$;
    eliminate the innermost $Q$ block;
    simplify $\phi$;
  od;
    compute input $\alpha$ for $SAT$ from $\phi$;
    invoke $SAT$ with the input $\alpha$;
end
```

Figure 7.1: The QUBOS Main Loop

case, in general miniscoping can only partially succeed in pushing the quantifier $\exists O_1, \ldots, O_p$ down in Φ; this in turn limits the miniscoping of the other quantifiers, *e.g.*, $\forall C_1, \ldots, C_n$. As a result, even after miniscoping, the average $\forall$-weight is

$$\frac{n+p+\mid \Phi \mid}{n+m+p+\mid \Phi \mid} = 1 - \frac{m}{m+n+p+\mid \Phi \mid}$$

which is high, up to 90%, for large n, m, p, and $\mid \Phi \mid$. The average $\exists$-weight tends to be better since by pushing down, even partially, the $\exists O_1, \ldots, O_p$, we increase the amount of ($\exists$-)structure in $\mathcal{P}$ and we obtain better average weight, typically between 50% and 70%. Furthermore, the average $\exists$-weight generally becomes larger (respectively smaller) when we decrease (respectively increase) one of the factors p and $\mid \Phi \mid$. Hence conditional planning gives us a potentially large spectrum of problems with differing amounts of structure. Moreover, there are standard databases of such planning problems that exhibit such variations, which we can use for testing.

7.4 QUBOS

We present in this section the decision procedure implemented by our system QUBOS. The main idea is to iterate normalization using miniscoping with selective quantifier expansion and simplification. For well-structured problems, the combination often does not require significant additional space; we will provide experimental evidence for this thesis in Section 7.5.

The structure of the main routine of our decision procedure is given in Figure 7.1. It takes as arguments a quantified Boolean formula ϕ and a SAT-solver SAT. The initial step determines whether the average quantifier weight is smaller for $\forall$ or $\exists$. Afterwards QUBOS iterates three transformations to reduce ϕ to a Boolean formula. As each iteration results in fewer Q-quantifiers, the procedure always terminates (given sufficient memory). At the end of this step, the formula ϕ contains only one kind of quantifier. Afterwards, QUBOS computes the input formula of the SAT-solver SAT depending on the quantifier kind Q and whether SAT operates on Boolean formulae or on formulae in clausal form. If Q is the quantifier $\exists$ then QUBOS deletes all the occurrences of Q and generates the input of SAT. If Q is the quantifier $\forall$ then QUBOS also deletes all the occurrences of Q, negates the resulting formula, generates the input of SAT, and finally it complements the result returned by the SAT solver.

Below, we describe the transformations used in the main loop in more details.

Miniscoping. Miniscoping is the process of pushing quantifiers down inside a formula to their minimal possible scope. By reducing the scope of quantifiers, miniscoping reduces the size of the formula resulting from subsequent quantifier expansion. The following rules for miniscoping are standard.

$$\begin{array}{lcll}
\forall x.\, \phi \wedge \psi & \Rightarrow & (\forall x.\, \phi) \wedge \forall x.\, \psi & \\
\forall x.\, \phi \vee \psi & \Rightarrow & (\forall x.\, \phi) \vee \psi, & \text{if } x \notin \mathsf{free}(\psi) \\
\forall x.\, \phi & \Rightarrow & \phi, & \text{if } x \notin \mathsf{free}(\phi) \\
\exists x.\, \phi \vee \psi & \Rightarrow & (\exists x.\, \phi) \vee \exists x.\, \psi & \\
\exists x.\, \phi \wedge \psi & \Rightarrow & (\exists x.\, \phi) \wedge \psi, & \text{if } x \notin \mathsf{free}(\psi) \\
\exists x.\, \phi & \Rightarrow & \phi, & \text{if } x \notin \mathsf{free}(\phi)
\end{array}$$

Note that similar kinds of simplification are performed in first-order theorem proving, where quantifiers are pushed down to reduce dependencies and generate Skolem functions with minimal arities (see [NW01]). Although simple and intuitively desirable, other QSAT solvers work by *maxi*scoping, *i.e.*, moving quantifiers outwards when transforming formulae to quantified clausal normalform.

Elimination of Quantified Variables. We explain only the elimination of universally quantified variables as the elimination of existentially quantified variables is similar. In an expansion phase, we eliminate blocks of universally quantified variables by replacing subformulae of the form $\forall x.\, \phi$ with the conjunction $\phi[\mathsf{true}/x] \wedge \phi[\mathsf{false}/x]$. In special cases (when eliminating universally quantified variables), we can avoid duplication altogether, *e.g.*, when ϕ does not contain existential quantifiers (*cf.*, [BL94]). In this case, we proceed as follows: we transform ϕ into the clausal normalform ψ, remove all tautologies from ψ, and then replace each literal from $\{y \mid y \text{ is universally quantified in } \phi\} \cup \{\neg y \mid y \text{ is universally quantified in } \phi\}$ with false in ψ.

Simplification. The application of simplification after each expansion step is important in keeping the size of formulae manageable. We distinguish between four kinds of simplification rules. The first kind consists of the standard simplification rules for Boolean logic that are used to remove tautologies, or perform direct simplification using the idempotence of the connectives $\vee$ and $\wedge$ and the fact that false and true are their (respective) identities.

The second kind of simplification rule is based on a generalization of the unit clause rule (also called Boolean constraint propagation [ZM88]). These rules are as follows (where l is a literal):

$$l \vee \phi \;\Rightarrow\; l \vee \phi[\mathsf{false}/l]$$

$$l \wedge \phi \;\Rightarrow\; l \wedge \phi[\mathsf{true}/l]$$

These rules are especially useful in combination with miniscoping as they often lead to new opportunities for miniscoping to be applied. For example, using the above rules, the formula

$$\forall x.\, \exists y, z.\, x \vee (y \wedge \neg z) \vee (\neg y \wedge z \wedge \neg x)$$

can be simplified to

$$\forall x.\, \exists y, z.\, x \vee (y \wedge \neg z) \vee (\neg y \wedge z)\,,$$

which can be further transformed using the miniscoping rules to

$$(\forall x.\ x) \vee ((\exists y.\, y) \wedge (\exists z.\, \neg z) \vee (\exists y.\, \neg y) \wedge (\exists z.\, z))\,. \tag{7.1}$$

This example also motivates why miniscoping is in the QUBOS main loop, as opposed to being applied only once initially.

The third kind of simplification rule consists of the following quantifier specific rules.

$\exists x.\, \phi$	$\Rightarrow$	ϕ,	if $x \notin \mathsf{free}(\phi)$
$\exists x.\, l$	$\Rightarrow$	true,	for $l \in \{x, \neg x\}$
$\exists x.\, x \wedge \phi$	$\Rightarrow$	$\phi[\mathsf{true}/x]$	
$\exists x.\, (\neg x) \wedge \phi$	$\Rightarrow$	$\phi[\mathsf{false}/x]$	
$\forall x.\, \phi$	$\Rightarrow$	ϕ,	if $x \notin \mathsf{free}(\phi)$
$\forall x.\, l$	$\Rightarrow$	false,	for $l \in \{x, \neg x\}$
$\forall x.\, x \vee \phi$	$\Rightarrow$	$\phi[\mathsf{false}/x]$	
$\forall x.\, (\neg x) \vee \phi$	$\Rightarrow$	$\phi[\mathsf{true}/x]$	

These rules are often effective in eliminating both kinds of quantifiers and therefore avoiding expansion steps. The application of these rules to the formula (7.1) above simplifies it to true.

The fourth kind of simplification rule is based on a technique commonly used by solvers based on clausal normalform and consists of dropping variables that occur only positively or only negatively in the clauses set. This technique can be also applied to quantified Boolean formulae that are in *nnf*. Let ϕ be a quantified Boolean formula in *nnf* and x a variable occurring in ϕ; we say that x is *monotone* in ϕ if it occurs only positively or only negatively in ϕ. It is easy to show that formulae with monotone variables have the following property.

Proposition 7.4.1 *Let ϕ be a quantified Boolean formula in* nnf *and let $Qx.\psi$ (for $Q \in \{\forall, \exists\}$) be a subformula in ϕ where x is monotone in ϕ. Then the formulae ϕ and ϕ' are equivalent, where:*

(i) If Q is the quantifier $\exists$ then ϕ' is obtained from ϕ by replacing $Qx.\psi$ with $\psi[\mathsf{true}/x]$ (respectively $\psi[\mathsf{false}/x]$), if x occurs positively (respectively negatively).

(ii) If Q is the quantifier $\forall$ then ϕ' is obtained from ϕ by replacing $Qx.\psi$ with $\psi[\mathsf{false}/x]$ (respectively $\psi[\mathsf{true}/x]$), if x occurs positively (respectively negatively).

This proposition provides a way of eliminating both universally and existentially quantified variables without applying the expansion step, provided the variables are monotone.

Clausal Normalform. Before handing off the normalized formula to a SAT solver we must transform it into clausal normalform. We do this using the renaming technique of [Boy92] where subformulae are replaced with new Boolean variables and definitions of these new Booleans are added to the formula. This technique allows the generation of the clauses in time linear in the size of the input formula.

7.5 Experimental Results

We have built a system, QUBOS (QUantified BOolean Solver), based on the ideas presented in Section 7.4. The system is written in C++ and supports the use of different SAT-solvers including PROVER [Sta89], HEERHUGO [GW00], SATO [Zha97] and ZCHAFF [MMZ$^+$01]. The times reported below are based on ZCHAFF. In these timings, typically 60% of the time is consumed by our system and 40% by ZCHAFF.

We carried out comparisons with the QBF [Rin01] and SEMPROP [Let01] systems, which are both state-of-the-art systems based on extensions of Davis-Putnam. The runtimes (on a 750 Mhz Sun Ultra Sparc workstation) depicted in the tables below are *user time* (in seconds) reported by the operating system for all computation required. Times greater than one hour are indicated by the symbol abort.

We used two sets of benchmarks for our comparison. The first is obtained by applying bounded model construction to a library of monadic formulae modeling several verification tasks. These problems include:

1. Formulae encoding the equivalence of the specification and implementation of a ripple-carry adder for different bit-widths.
2. Formulae stating safety properties of a lift-controller.
3. Formulae encoding the equivalence of von Neumann adders and ripple-carry adders with varying bit-width.
4. Formulae stating the stability of a timed flip-flop model.
5. Formulae stating the mutual exclusion property for two protocols.

The second set contains encodings of conditional planning problems generated by Rintanen [Rin01] as well as their negations.

Tables (7.1-7.6) show the results of the comparison. Each table gives information on quantificational structure, the size k of the model investigated, running times, QUBOS space requirements in megabytes, the average quantifier width, and the prefix type of the problems. The input formulae are of size 10^5, on average, with respect to $|\,.\,|$ defined in Section 7.2. QUBOS has dramatically better performance on all of these examples. The reason is that these problems all have very high structure and, as explained previously, the amount of structure improves (the average quantifier weight decreases) as k and the formulae become larger. These examples also demonstrate that, for well-structured formulae, memory requirements are typically modest; for example, the adder problems use 2 megabytes on the average. On the other hand, QBF and SEMPROP translate the problems into quantified clausal form, which drastically increases the quantifier scope and the time and space required to find a solution.

The second set of examples contains encodings of block-world planning problems where there is significantly less structure, although varied. Table 7.7 and Table 7.8 show the time required to solve different block planning problems and their negations. The instances are called $x.iii.y$, where x denotes the number of blocks, y denotes the length of the plan and iii stands for the encoding strategy used to generate the problem (*cf.*, [Rin99b]). The instances are ordered by the number of the

k	Qbf	Semprop	Qubos		
	time	time	time	space	$aw_\forall\%$
invalid, Σ_3, $Q \equiv \forall$					
1	0	0	0	0	20
2	54	0	0	0	12
3	abort	661	0	0	7
4	abort	abort	0	0	5
5	abort	abort	0	0	3
6	abort	abort	2	6	2.8
7	abort	abort	18	11	2.2
8	abort	abort	59	20	1.6
9	abort	abort	445	38	1.8
10	abort	abort	1945	74	1.3
valid, Π_4, $Q \equiv \exists$					
1	0	0	0	0	32
2	25	0	0	0	17
3	abort	39	0	0	10
4	abort	abort	2	7	6
5	abort	abort	10	27	4

Table 7.1: Ripple-carry Adder

k	Qbf	Semprop	Qubos		
	time	time	time	space	$aw_\forall\%$
(invalid, Π_2)					
8	abort	20	0	0	11
16	abort	abort	0	0	5
32	abort	abort	0	0	2
64	abort	abort	0	0	1
128	abort	abort	1	0	0.7

Table 7.2: Mutual Exclusion Protocol

blocks and their size. Table 7.7 contains the results of the (positive) block planning problems and Table 7.8 contains the results of the negated block planning problems. A (positive) block planning problem has the general form $\exists\forall\exists\phi$, where ϕ is a Boolean formula, and its negation has the form $\forall\exists\forall\neg\phi$. Since the negative problems are just the negation of the positive problems the average $\exists$-weight in the positive

k	QBF	SEMPROP	QUBOS		
	time	time	time	space	$aw_Q\%$
invalid, Σ_2, $Q \equiv \forall$					
1	0	0	0	0	21
2	0	0	0	0	8
4	2	0	0	0	6
8	20	abort	1	0	4
16	139	abort	6	9	3
32	abort	abort	24	16	2.8
64	abort	abort	150	29	2.4
128	abort	abort	1325	55	2
valid, Π_3, $Q \equiv \exists$					
1	1	0	0	0	21
2	abort	0	0	0	8
3	abort	3	0	0	7
4	abort	238	0	0	6
5	abort	abort	0	0	5
6	abort	abort	0	0	4
7	abort	abort	1	0	3

Table 7.3: Lift Controller

k	QBF	SEMPROP	QUBOS		
	time	time	time	space	$aw_\forall\%$
(invalid, Σ_3)					
5	2	1	1	25	1
6	9	3	3	53	0.8
7	36	7	6	107	0.6
8	137	15	13	201	0.5
9	454	29	25	353	0.49
10	1318	54	44	586	0.43
11	abort	97	77	928	0.38
12	abort	167	134	1413	0.34

Table 7.4: FlipFlop

case and the average $\forall$-weight in the negative case are identical and their values are displayed in the second column of Table 7.7 and Table 7.8 respectively.

k	QBF	SEMPROP	QUBOS		
	time	time	time	space	$aw_\forall$%
(invalid, Σ_3)					
8	abort	22	13	105	0.06
9	abort	41	24	175	0.04
10	abort	73	41	280	0.03
11	abort	110	63	430	0.028
12	abort	172	98	640	0.022
13	abort	239	147	922	0.018
14	abort	386	218	1297	0.014
15	abort	558	311	1786	0.012

Table 7.5: Von Neumann Adders

k	QBF	SEMPROP	QUBOS		
	time	time	time	space	$aw_\forall$%
(invalid, Σ_3)					
4	abort	5	0	0	92
6	abort	abort	1	0	31
8	abort	abort	6	7	19
10	abort	abort	20	11	13
12	abort	abort	54	18	10
14	abort	abort	128	30	8
16	abort	abort	283	40	7
18	abort	abort	592	72	6
20	abort	abort	1064	107	5.7
22	abort	abort	1951	154	5
24	abort	abort	3153	215	4

Table 7.6: Szymanski Protocol

In the positive case, the system SEMPROP generally either diverges or is very fast. The system QBF always succeeds with respectable runtime. For QUBOS there is a close relationship between its success and the average quantifier weight: the performance of QUBOS decreases as the average quantifier weight rises. QUBOS succeeds for the small problems, up to size 10^3 (with respect to $|\,.\,|$), even when the average quantifier weight is high, but it requires significantly more time than QBF. When the problems become larger, up to size 10^5, and the average quantifier weight is high, then QUBOS exhausts memory. The superior performance of QBF in this

		Positive (∃∀∃)			
		QBF	SEMPROP	QUBOS	
instance	aw$_\exists$%	time	time	time	space
2.iii.2	69	4	0	0	0
2.iii.3	70	12	1	2	14
2.iii.4	71	24	118	4	19
2.iii.5	71	41	1512	6	25
2.iii.6	72	67	abort	9	30
2.iii.7	72	100	abort	15	35
2.iii.8	73	139	abort	19	42
2.iii.9	74	194	abort	23	47
2.iii.10	75	255	abort	26	53
3.iii.2	55	0	0	0	7
3.iii.3	58	0	0	1	12
3.iii.4	60	1	0	2	17
3.iii.5	61	2	abort	3	21
4.iii.2	60	6	0	25	157
4.iii.3	64	14	0	67	307
4.iii.4	67	25	abort	124	457
4.iii.5	67	41	abort	204	607
4.iii.6	69	61	abort	299	756
4.iii.7	70	84	abort	450	907
5.iii.2	54	115	1	579	1591
5.iii.3	61	226	2	1845	2034
5.iii.4	65	373	abort	abort	-
5.iii.5	67	561	abort	abort	-
5.iii.6	67	785	abort	abort	-
5.iii.7	70	1053	abort	abort	-
5.iii.8	72	1379	abort	abort	-

Table 7.7: Block-World Planning Problems

domain is not too surprising: it was developed and tuned precisely to solve this class of planning problems.

In the negative case, the results show that QUBOS is robust with respect to the quantificational structure and its success depends decisively on the average quantifier weight. Notice that although the problems in the positive case as well as in the negative case have the same average quantifier weight QUBOS requires in general less CPU time for the negative problems. This can be explained by the fact that

		Negative ($\forall\exists\forall\exists$)			
		QBF	SEMPROP	QUBOS	
instance	aw$_\forall$%	time	time	time	space
2.iii.2	69	abort	abort	1	3
2.iii.3	70	abort	abort	3	5
2.iii.4	71	abort	abort	8	7
2.iii.5	71	abort	abort	15	8
2.iii.6	72	abort	abort	25	9
2.iii.7	72	abort	abort	39	11
2.iii.8	73	abort	abort	58	12
2.iii.9	74	abort	abort	82	13
2.iii.10	75	abort	abort	115	14
3.iii.2	55	abort	abort	0	0
3.iii.3	58	abort	abort	0	0
3.iii.4	60	abort	abort	0	0
3.iii.5	61	abort	abort	0	0
4.iii.2	60	abort	abort	1	0
4.iii.3	64	abort	abort	3	5
4.iii.4	67	abort	abort	8	7
4.iii.5	67	abort	abort	15	8
4.iii.6	69	abort	abort	25	10
4.iii.7	70	abort	abort	40	11
5.iii.2	54	abort	abort	12	12
5.iii.3	61	abort	abort	47	15
5.iii.4	65	abort	abort	90	19
5.iii.5	67	abort	abort	180	24
5.iii.6	67	abort	abort	327	28
5.iii.7	70	abort	abort	468	32
5.iii.8	72	abort	abort	749	39

Table 7.8: Block-World Planning Problems

the negation makes these problems easier. When applying QBF and SEMPROP to the negative problems, the negated formula $\neg\phi$ is first transformed into clausal form and thereby a new block of existential quantified variables (due to the renaming technique describe in Section 7.4) is introduced and so these problems have a $\forall\exists\forall\exists$-structure. As a result these problems no longer have the shape of $\exists\forall\exists$ planning problems, which accounts for the divergence of QBF.

Notice that the MONA system can be also used for these examples. A detailed comparison of MONA with the MONACO is given in Chapter 6. On the examples given here MONA yields comparable results for the ripple-carry adder, flip-flop, and mutex examples. It yields poorer results for the von Neumann adders, lift-controller, and planning problems. For example, for the von Neumann adders with bit-width less than 11 it is up to factor 3 slower than QUBOS and it diverges on the rest the von Neumann adders, the lift-controller, and all of the planning problems.

7.6 Related Work

The idea of tuning a solver to exploit structure also arises in bounded model checking, where SAT-solvers are tuned to exploit the problem-specific structure arising there. In [Sht00], such heuristics were embedded within a generic SAT algorithm that generalizes the Davis-Putnam procedure. Similar techniques to miniscoping and quantifier expansion are also used in Williams et al. [WBCG00] to optimize different computation tasks like the calculation of fixed points.

Most QBL algorithms generalize the Davis-Putnam procedure to operate on formulae transformed into quantified clausal normal form. Cadoli et al. [CGS98] and Rintanen [Rin01, Rin99b] present different heuristic extensions of the Davis-Putnam method. Cadoli et al.'s techniques were tuned for randomly generated problems and Rintanen's strategies were specially designed for planning problems whose quantifiers have a fixed $\exists\forall\exists$-structure. Other work includes that of Letz [Let01] and Giunchiglia et al. [GNT01a] who have generalized the *backjumping* heuristic (also called dependency-directed backtracking) to QBL. Our approach differs from all of these in that it is not based on Davis-Putnam, it can operate freely on subformulae of the input formula (this avoids a major source of inefficiency of Davis-Putnam based procedures, namely the selection of branching variables is strongly restricted by the ordering induced by the prefix of the input formula), and for structured problems (in our sense) it yields significantly better results.

The most closely related work is that of Plaisted et al. [PBZ02] who present a decision procedure for QBL that also operates directly on quantified Boolean formulae by iteratively applying equivalence preserving transformation. However, rather than expanding quantifiers, in their approach a subformula with a set of free variables X is replaced by a large conjunction of all negated evaluations of X that make the subformulae unsatisfiable. Plaisted et al. [PBZ02] suggest that their procedure should work well for hardware systems that have structure in the sense of being "long and thin"; as indicated by their examples (ripple-carry adders), these systems form a subclass of well-structured problems in our sense. As no implementation is currently available, we were unable to compare our approaches experimentally.

7.7 Chapter Summary

We presented an approach to deciding quantified Boolean logic that works directly on fully-quantified Boolean formulae. We gave a characterization of structure, defined an interesting, natural, class of well-structured problems, and showed experimentally that our approach works well for problems in this class.

One issue that is not addressed in our implementation of QUBOS is the impact of the order in which quantified subformulae are expanded. Currently QUBOS selects the innermost quantified subformula. As future work, we intend to investigate the effect of different selection strategies, such as ordering the quantified formulae with respect to their relative structure.

Chapter 8

Conclusions and Further Research

As suggested by the title of the thesis, our primary goal is to investigate system verification based on monadic second-order logics. In this final chapter, we summarize the main contributions and results, and discuss future directions of research.

8.1 Summary

As the previous chapters illustrate, the use of formal methods to develop reliable and correct safety-critical systems is highly advisable and beneficial. In the last few years, industry has looked at formal methods with a constantly increasing interest. While powerful and useful, interactive and semi-automatic formal methods require both advanced knowledge in mathematics from the part of the verification engineer, and significant increase on the cost of system development, since proofs in these approaches tend to be intricate and time consuming. For automatic formal methods, on the other hand, the time costs to assist the verification process are minimal and there is no strong requirements on the skill of the verification engineer, only some familiarity with logic is necessary. Hence, from this point of view, industry gives a considerable importance to automatic verification methods, the "push-button" technology.

To make automatic formal methods more accessible for industry, we looked for logics in which verification tasks can be automatically achieved and allow for the verification of a large class of systems. For this reason, we focused on the monadic second-order logics, since they are among the most expressive decidable logics known. In the system verification approach based on monadic second-order logics, system description and system properties are both formulated directly within the language of the underlying logic whose decision procedure is used then to tackle the problem of automatically verifying that the modeled system satisfies the specified properties.

The main drawbacks of this approach are the low-level languages that this kind of logics provide for modeling and specifying and the high computational complexity of their decision procedures. In this thesis, we proposed several approaches that address these limitations. Our contributions can be highlighted using the keywords: LISA, BMC, MONACO, QUBOS, and embedding LTL model checking in MONA. In the following, we briefly review these contributions.

LISA is the answer to the questions how to increase the expressive power of the existing specification languages and provide a handle on their computational complexity. LISA is equipped with a datatype declaration system, which not only allows for the declaration of types in a high-level way that is close to that of conventional programming languages but also offers a mechanism that helps users to estimate the complexity of the verification tasks. The first experimental results of the prototypical implementation of LISA encourage us to go further in this direction and to add other high-level constructors that help structuring specifications.

The LISA implementation is coupled with the MONA system in the following way. LISA specifications are compiled into WS2S formulae whose validity is then decided using MONA. Because of that the performance of LISA is strongly dependent on the performance of MONA. Since MONA suffers from the state-space explosion problem, we investigated this weakness and proposed the bounded model construction (BMC) approach as an alternative. We explored the BMC problem for a series of monadic logics on finite words as well as infinite words, and have obtained theoretical and practical results. The theoretical contributions are twofold: (i) we established the BMC complexity results for the logics M2L-STR, WS1S, S1S and their first-order fragments, and (ii) we used the insights we have gained from these results, to shed some light on the differences between the logics M2L-STR and WS1S, and to prove that no elementary validity preserving translation of WS1S into M2L-STR exist, and thereby answered a question that remained open for long time. The practical contributions regard the logic M2L-STR. We have obtained a procedure for generating counter-examples that is non-elementary faster than the standard automata-based decision procedures. We implemented this procedure in the MONACO system and demonstrated, for a large suite of examples taken from diverse domains, that MONACO provides a more efficient alternative to MONA.

The application of MONACO to the bytecode verification problem, discussed in Chapter 6, shows that the bounded model construction for M2L-STR can successfully be used in a complementary way to other approaches like on-the-fly model checking for LTL (implemented in SPIN) and symbolic model checking for CTL (implemented in SMV). Using the MONACO system we were able to discover errors in examples of buggy bytecode, which was not possible using SPIN and SMV. The system MONACO supports the use of multiple state-of-the-art QSAT-solvers: the choice of the solver to be used as backend has an enormous impact on the performance of MonaCo. An advantage of this is that any optimization and improvement of a backend necessarily leads to an improvement on the performance of MonaCo.

The poor practical results obtained by using the existing Davis-Putnam based QSAT-solvers as backend of MonaCo have motivated our work on the Qubos procedure, which is based on combining techniques like miniscoping, quantifier expansion, and eager simplification, and works directly on fully-quantified Boolean formulae. We gave a characterization of structure, defined an interesting, natural, class of well-structured problems, and showed that our approach works well for problems in this class and even outperforms other QBL solvers.

In the investigation of BMC for monadic second-order logics on infinite words, we faced the problem of using finite words to represent models that consists of infinite words. We solved this problem by adapting the notion of lasso-words. The notion of lasso-words is also the central idea behind our encoding of LTL model checking in M2L-Str. Besides for the practical use of this encoding itself, we obtained several theoretical results. First, the encoding opens up the possibility to formalize liveness properties (properties interpreted over infinite words/computations) and decide them using the decision procedure for M2L-Str, a fact that seemed a priori impossible since this logic handles finite words. Second, our encoding can be seen as a new LTL automata-based model checker in which finite automata are used instead of Büchi automata, and this not only avoids the problem of complementing Büchi automata but also offers access to the automata constructions for finite words which are simpler as is the case for Büchi automata. Moreover, it allows for the use of the automaton minimization operation, which is not available for Büchi automata.

8.2 Future Work

Our results leave room for improvements and extensions. In the following, we discuss work still to be done on both the practical and on the theoretical side for each of our contributions.

Lisa

The experiments carried out using the prototypical implementation of Lisa showed that the integrated datatype declaration system has a great impact on the readability and structuring of specifications. Nevertheless, we can say that Lisa is still a primitive specification language and lacks of more high-level constructors that could allow, for example, the decomposition of specifications into modules that support abstraction and specification reuse. On the practical side, the Lisa system needs significant improvement. In Section 3.7.2, we discussed the concept of guided trees used by MONA to efficiently construct tree automata. Guided trees are not used in the current LISA implementation and could be a subject of future investigation. On the experimental side, there is still work to do. Up to now we focused only on protocol verification, and in the future, one could tackle problems stemming from linguistic applications.

BMC

A question arises for what concerns BMC is whether the complexity results for monadic second-order logics on words can be transfered to monadic second-order logics on trees. We believe that this question can be answered straightforwardly. For the logics WS2S and S2S we expect that the same results as for WS1S and S1S respectively hold, *i.e.* BMC for these logics is non-elementary. This is because the logics WS1S and S1S can be encoded in WS2S and S2S respectively using polynomial translations. We also believe that the same results as for M2L-STR can be established for M2L-TREE. With minor changes, the translation $\lceil . \rceil_k$ given in Section 4.2.1, which maps an M2L-STR formula and a natural number k into a quantified Boolean formula, can be adapted for M2L-TREE formulae. All these points have to be carefully explored in detail.

The LTL Embedding in M2L-Str

Here we see a considerable amount of further work. On the practical side, to investigate the usefulness of our encoding of LTL model checking in M2L-STR, we have to implement it and carry out experiments. Technically, all that we need for this purpose is to implement the translation $[\![.]\!]$ (*cf.* Section 5.3.4), which maps LTL formulae and finite-state systems into M2L-STR formulae, and to couple this with the MONA system. The positive experimental results of the encoding of LTL satisfiability in MONA reported in [HH01] give us hope that similar positive results for the encoding of LTL model checking in MONA can be obtained.

On the theoretical side, we still have to provide an analysis of the computational complexity of $[\![.]\!]$. We believe that the automata constructions performed by MONA to decide the formulae yielded from model checking problems do not require more amount of time and space as the time and space amount required by other LTL model checkers.

QUBOS

One issue that is not addressed in the implementation of QUBOS is whether its efficiency depends on the order in which quantified subformulae are expanded. Currently Qubos selects the innermost quantified subformula. To investigate the effect of different selection strategies, such as ordering the quantified formulae with respect to their relative structure, would be an interesting research topic of future work.

Appendix A

Background on ω-languages

A.1 Proof of Lemma A.1.2

Let M and L be two ω-regular languages. Based on Proposition 4.3.2, we reduce the language containment problem $M \subseteq L$ to a *lasso containment problem.*

Proposition A.1.1 *Let M and L be two ω-regular languages, then $M \subseteq L$ iff for all $\pi \in \Sigma^\omega$, if π is lasso and $\pi \in M$, then $\pi \in L$.*

Furthermore, we also reduce the lasso containment problem to a *bounded lasso containment problem.* That is, in instead of checking that all lassos $\pi \in M$ are in L, it suffices to check that only lassos $\pi \in M$ of bounded length are in L. In order to do this, we establish the following property.

Lemma A.1.2 *Let $\mathcal{A}$ be a Büchi automaton. The language $L(\mathcal{A})$ is not empty if and only if $\mathcal{A}$ accepts a lasso word π of length $k \leq |\mathcal{A}|$.*

Proof The "if" condition is trivial. We prove the "only if" condition. Assume $L(\mathcal{A})$ is not empty. By Proposition 4.3.2, $L(\mathcal{A})$ contains a (l, k)-lasso word. Let $q_0 \dots q_{l-1}(q_l \dots q_{k-1})^\omega$ be its accepting run. Without loss of generality, we assume that the subsequences $q_0 \dots q_{l-1}$ and $q_l \dots q_{k-1}$ contain no loops. Now, if $k \leq |\mathcal{A}|$ then we are done. In the other case, let i be a minimal natural number with $i < l$ and such that there is a natural number j with $l \leq j < k$ and such that q_i and q_j are identical.

$$\begin{tikzcd}
q_0 \arrow[r, dotted] & q_i \arrow[r, dotted] & q_{l-1} \arrow[r] & q_l \arrow[r, dotted] & q_j \arrow[r, dotted] & q_{k-1} \arrow[ll, bend right]
\end{tikzcd}$$

Now, we can shorten the above run as follows:

$$\begin{tikzcd}
q_0 \arrow[r, dotted] & q_i \arrow[r] & q_{j+1} \arrow[r, dotted] & q_{k-1} \arrow[r] & q_l \arrow[r, dotted] & q_{j-1} \arrow[llll, bend right]
\end{tikzcd}$$

By construction, the run $q_0 \dots q_{i-1}(q_i q_{j+1} \dots q_{k-1} q_l \dots q_{j-1})^w$ is accepting and contains no loops and thus its length is smaller than or equal to $|A|$. ■

Using the above lemma, we reformulate Proposition A.1.1 as follows:

Lemma A.1.3 *Let M and L be two ω-regular languages and $\mathcal{A}$ an automaton accepting the language $M \cap \overline{L}$, then $M \subseteq L$ iff for all lasso words π with length smaller than $|\mathcal{A}|$, if $\pi \in M$, then $\pi \in L$.*

Appendix B

Additional Proof Details of Theorem 4.3.19

B.1 Some Properties of $\langle . \rangle_k^l$

We establish some useful properties of the function $\langle . \rangle_k^l$ (Definition 4.3.8) and lasso substitutions (Definition 4.3.10).

Lemma B.1.1 *Let k, l be natural numbers with $l < k$, σ be an (l,k)-lasso substitution, and X be a second-order variable.*

(1) $m \equiv \langle m \rangle_k^l \pmod{k-l}$, *for all* $m \in \mathbb{N}$.

(2) If $m \equiv n \pmod{k-l}$, *then* $n \in \sigma(X)$ *iff* $m \in \sigma(X)$, *for all* $m, n \in \mathbb{N}$.

(3) $\langle i \rangle_k^l = k - 1$ *iff* $\langle i+1 \rangle_k^l = l$, *for all* $i \in \mathbb{N}$.

(4) $\langle i \rangle_k^l < k - 1$ *iff* $\langle i+1 \rangle_k^l = 1 + \langle i \rangle_k^l$, *for all* $i \in \mathbb{N}$.

(5) If σ is a (l_1, k_1)-lasso substitution and M is a (l_2, k_2)-lasso set, then $\sigma[M/X]$ is a (l,k)-lasso substitution where $l = \mathsf{lcm}(k_1, k_2)$ and $k = l + \mathsf{lcm}(k_1 - l_1, k_2 - l_2)$, where lcm *is the least common multiple of m and n.*

Proof The facts (1-4) are obvious. For (5), we have to prove that for the set M and the set $\sigma[M/X](Y)$, for Y in domain of σ, are (l,k)-lasso. This means we have to show that forall $n \in \mathbb{N}$, it holds:

$$\langle n \rangle_{k_2}^{l_2} \in M \text{ iff } \langle \langle n \rangle_k^l \rangle_{k_2}^{l_2} \in M$$
and
$$\langle n \rangle_{k_1}^{l_1} \in \sigma(Y) \text{ iff } \langle \langle n \rangle_k^l \rangle_{k_1}^{l_1} \in \sigma(Y), \text{ for } Y \text{ in domain of } \sigma\,.$$

To achieve this, it suffices to prove that the equations $\langle \langle n \rangle_k^l \rangle_{k_1}^{l_1} = \langle n \rangle_{k_1}^{l_1}$ and $\langle \langle n \rangle_k^l \rangle_{k_2}^{l_2} = \langle n \rangle_{k_2}^{l_2}$ hold. We establish only the first equation the second one follows similarly. Suppose $n < l_1$, then it follows that $n < l$ and thus $\langle n \rangle_{k_1}^{l_1} = \langle \langle n \rangle_k^l \rangle_{k_1}^{l_1} = n$. Now suppose $n \geq l_1$, then by Definition 4.3.10,

$$\begin{aligned} \langle \langle n \rangle_k^l \rangle_{k_1}^{l_1} &= \langle l + (n-l) \mathsf{~mod~} (k-l) \rangle_{k_1}^{l_1} \\ &= l_1 + ((l + (n-l) \mathsf{~mod~} (k-l)) - l_1) \mathsf{~mod~} (k_1 - l_1) \end{aligned} \tag{B.1}$$

Now from the equation $k = l + \mathsf{lcm}(k_1 - l_1, k_2 - l_2)$, it follows that there is some $q \in \mathbb{N}$ with $k - l = q(k_1 - l_1)$ and thus we have

$$(n-l) \mathsf{~mod~} (k-l) = (n-l) - q'(k_1 - l_1), \text{ for some } q' \in \mathbb{N}.$$

The proof follows by simplifying (B.1) using the above equation and unfolding Definition 4.3.8. ■

B.2 Proof of Lemma 4.3.12

Proof We proceed by induction over the structure of the formula ϕ. We first establish the claim for the atomic formula $X(t)$. The claim can be reduced to prove that $\sigma(t) \in \sigma(X)$ and $\langle \sigma(t) \rangle_k^l \in \sigma(X)$ are equivalent. This holds by combining (1) and (2) of Lemma B.1.1.

In the induction step we consider only the cases where ϕ is of the form $\exists p.\, \phi'$ and $\exists X.\, \phi'$ as the remaining cases are straightforward. Suppose that ϕ is of the form $\exists p.\, \phi'$. By the definition of the original semantics of S1S $\sigma \models \exists p.\, \phi'$ is equivalent to $\sigma[n/p] \models \phi'$ for some $n \in \mathbb{N}$. Because σ is a (l,k)-lasso, it follows by the definition of lasso substitution that the substitution $\sigma[n/p]$ is also (l,k)-lasso. Now, by the induction hypothesis it follows that $\sigma[n/p] \models \phi'$ is equivalent to $\sigma[n/p] \models_k^l \phi'$ and this is equivalent to $\sigma \models_k^l \exists p.\, \phi$ by the definition of the bounded semantics.

Consider now the case where ϕ is of the form $\exists X.\, \phi'$. By the definition of the original semantics of S1S $\sigma \models \exists X.\, \phi'$ is equivalent to $\sigma[M/X] \models \phi'$ for some $M \subseteq \mathbb{N}$.

By (5) of Lemma B.1.1 there is some set M' such that the substitution $\sigma[M'/X]$ is a (l', k')-lasso for some $l', k' \in \mathbb{N}$ with $l' < k'$ and such that $\sigma[M'/X] \models \phi'$. By applying the induction hypothesis, we conclude that $\sigma[M'/X] \models \phi'$ is equivalent to $\sigma[M'/X] \models^{l'}_{k'} \phi'$ and this is again equivalent to $\sigma \models^{l}_{k} \exists X.\, \phi'$. ■

B.3 Proof of Lemma 4.3.14

Proof (i): We use following abbreviations: δ for $\sigma[n/p]$ and δ' for $\sigma[\langle n\rangle^{l}_{k}/p]$. We proceed inductively over the construction of t. In the base case t is the variable p. It follows, $\delta(t) = n$ and $\delta'(t) = \langle n\rangle^{l}_{k}$. The aim follows by Lemma B.1.1(1). In the step case t is of the $t'{+}1$; in this case we have $\delta(t'{+}1) = 1{+}\delta(t)$ and $\delta'(t'{+}1) = 1{+}\delta'(\langle n\rangle^{l}_{k})$. By induction, $\delta(t') \equiv \delta'(t') \;(\mathsf{mod}\,(k-l))$ and thus, $\delta(t) \equiv \delta'(t) \;(\mathsf{mod}\,(k-l))$.

(ii): We also proceed by induction over the structure of the formula ϕ. The atomic formula $X(t)$ follows immediately from (i). The inductive case $\exists q.\, \phi'$ holds straightforwardly by applying the induction hypothesis. ■

B.4 Proof of Lemma 4.3.16

Proof We proceed by induction over the structure of ϕ. In the induction basis we consider the atomic formula $X(t)$. By the definition of the bounded semantics, it holds $\sigma \models^{l}_{k} X(t)$ iff

$$\langle\sigma(t)\rangle^{l}_{k} \in \sigma(X_i). \tag{B.2}$$

Observe that $\lceil X(t)\rceil = \exists p.\, \mathsf{lasso}(l, k, t, p) \wedge p \in X$. By the definition of the semantics of M2L-Str, it holds $\sigma^{k} \models_{\text{M2L}} \exists p.\, \mathsf{lasso}(l, k, t, p) \wedge p \in X$ iff

$$\sigma^{k}[n/p] \models_{\text{M2L}} \mathsf{lasso}(l, k, t, p) \wedge p \in X, \text{ for some } n \leq k. \tag{B.3}$$

By choosing n in (B.3) as $\langle\sigma(t)\rangle^{l}_{k}$ and using Lemma B.1.1 it follows that (B.2) and (B.3) are equivalent.

In the induction step we consider only the case where ϕ is of the form $\exists p.\, \phi'$ as the remaining cases are trivial. By the definition of the bounded semantics, we have $\sigma \models^{l}_{k} \exists p.\, \phi'$ iff $\sigma[n/p] \models^{l}_{k} \phi'$ for some $n \in \mathbb{N}$. By Lemma 4.3.14, $\sigma[n/p] \models^{l}_{k} \phi'$ iff

$$\sigma[\langle n\rangle^{l}_{k}/p] \models^{l}_{k} \phi'. \tag{B.4}$$

By applying the induction hypothesis, it follows that (B.4) is equivalent to

$$(\sigma[\langle n\rangle_k^l/p])^k \models_{\text{M2L}} \lceil \phi' \rceil_k^l. \tag{B.5}$$

By the semantics of M2L-Str (B.5) is equivalent to $\sigma^k \models_{\text{M2L}} \lceil \exists p.\, \phi' \rceil_k^l$. ■

Appendix C

Code

C.1 Specificatition of a Lift System in LLISA

```
data Etage  = 0..3;
data Level  = record {set:bool, reset:bool,req:bool};
data Levels = array Etage of Level;
data Goal   = 0..4;
data Dir    = up,down,none;
data Door   = open,opening,closed,closing;
data Ppd    = nobody, somebody,vip;
data State = record {levels:Levels,
                     pos:Etage,
                     goal:Goal,
                     dir :Dir,
                     door:Door,
                     ppd:Ppd};

data Lift  = array nat of State;

pred init_level (l:Level) = l.req ;

pred init(s:State) =
        (all Etage p: init_level((s.levels)[p])) &
        s.pos   = 1    &
        s.goal  = 4    &
        s.dir   = none &
        s.door  = open &
        s.ppd   = nobody;

pred stop(s:State) =
   (ex Etage p: s.pos = p & (s.levels)[p].req) |
   (s.goal = s.pos)                             |
   s.pos = 0 & s.dir  = down                    |
   s.pos = 3 & s.dir  = up;
```

```
pred extern_up_req(s:State)   =
        ex Etage p: p ~=0 & (s.levels)[p].req & s.pos < p;

pred extern_down_req(s:State) =
        ex Etage p: p ~= 3 & (s.levels)[p].req & s.pos > p;

pred go_up(s:State)= s.goal ~= 4 & s.goal > s.pos ;

pred go_down(s:State) = s.goal ~= 4 & s.goal < s.pos ;
pred next_door(s,t:State) =
  if stop(s)
     then case s.door of
             closed   => t.door = opening;
             opening  => t.door = open;
             open     => t.door = open;
             closing  => t.door = closing;
          esac
   else case s.door of
             open    => t.door = closing;
             closing => t.door = closed;
             opening => t.door = opening;
             closed  => t.door = closed;
          esac;
pred next_pos(s,t:State) =
   if stop(s)
     then
        case s.dir  of
             up   => t.pos = (s.pos) + 1;
             down => t.pos = (s.pos) - 1;
             none => t.pos = s.pos;
        esac
   else t.pos = s.pos;

pred next_ppd(s,t:State) =
   cond
     s.door = open => (if s.pos = 0 then (t.ppd = nobody | t.ppd = vip));
                 => t.ppd = s.ppd;
   dnoc;

pred next_goal(s,t:State) =
  cond
     s.pos = s.goal& s.door = opening => t.goal = 4;
     s.door = open & s.ppd ~= nobody  => case s.ppd of
                                           somebody => 0 ~= t.goal &
                                                       t.goal ~= 4;
                                           vip      => t.goal ~= 4;
                                         esac;
     => t.goal = s.goal;
  dnoc;

pred right_dir(s:State) = ((extern_up_req(s) | go_up(s)) & (s.dir = up)) |
                          ((extern_down_req(s) | go_down(s)) &
                           (s.dir = down));
```

```
pred next_dir(s,t:State) =
  cond
    stop(s) | s.door ~= closed => t.dir = none;
    right_dir(s)               => t.dir = s.dir;
    s.dir = none =>  cond
                       go_up(s)   => t.dir = up;
                       go_down(s) => t.dir = down;
                       extern_up_req(s)   => t.dir = up;
                       extern_down_req(s) => t.dir = down;
                                          => t.dir = none;
                     dnoc;
      => t.dir = none;
  dnoc;

pred next_req_level (a,b:Level) =
    cond
      a.set   => b.req;
      a.reset => ~ b.req;
              => a.req <-> b.req;
    dnoc;

pred next_req(s,t:State) =
        all Etage p: next_req_level((s.levels)[p],(t.levels)[p]);

pred next (s,t:State) =
         next_req(s,t)&
         next_door(s,t) &
         next_ppd(s,t)  &
         next_goal(s,t) &
         next_pos(s,t)  &
         next_dir(s,t);

pred run (L:Lift) = init(L[0]) & all nat i: 0<i -> next(L[i-1],L[i]);

pred spec(L:Lift) = all nat i: L[i].dir ~=none -> L[i].door =closed;

var  Lift L;
run(L) -> spec(L);
```

C.2 The encoding of the Lift system in Mona

```
#LLisa v0.1-BETA
#Copyright (C) December 2002 University of Freiburg
#Mona-Code genereted automatically

m2l-str;

pred is_bool(var0 @p0@) = true;
pred bool_eq(var0 @p0@, @p1@) = @p0@ <=> @p1@;
pred is_nat(var1 @p2@) = true;
pred nat_eq(var1 @p2@, @p3@) = @p2@ = @p3@;
pred nat_less(var1 @p2@, @p3@) = @p2@ < @p3@;
pred nat_lesseq(var1 @p2@, @p3@) = @p2@ <= @p3@;
pred is_Etage(var0 @p4@0, @p4@1) = true;
pred Etage_eq(var0 @p5@$0_1, @p5@$1_1, var0 @p5@$0_2, @p5@$1_2) =
  (@p5@$0_1 <=> @p5@$0_2) & (@p5@$1_1 <=> @p5@$1_2);

pred is_Etage_0(var0 @p13@0, @p13@1) = ~@p13@0 & ~@p13@1;
pred is_Etage_1(var0 @p14@0, @p14@1) = @p14@0 & ~@p14@1;
pred is_Etage_2(var0 @p15@0, @p15@1) = ~@p15@0 & @p15@1;
pred is_Etage_3(var0 @p16@0, @p16@1) = @p16@0 & @p16@1;
pred Etage_add(var0 @p6@$0, @p6@$1, var0 @p7@$0, @p7@$1, var0 @p8@$0, @p8@$1) =
  (@p8@$0 <=> ~(@p6@$0 <=> ~(@p7@$0 <=> false))) &
  (@p8@$1 <=>
   ~(@p6@$1 <=> ~(@p7@$1 <=> (false & ~(@p6@$0 <=> @p7@$0) | @p6@$0 & @p7@$0))))
  &
  ~((false & ~(@p6@$0 <=> @p7@$0) | @p6@$0 & @p7@$0) & ~(@p6@$1 <=> @p7@$1) |
    @p6@$1 & @p7@$1);

pred Etage_less(var0 @p9@$0, @p9@$1, var0 @p10@$0, @p10@$1) =
  (~@p9@$1 & @p10@$1 |
   (@p9@$1 <=> @p10@$1) & (~@p9@$0 & @p10@$0 | (@p9@$0 <=> @p10@$0) & false)) &
  (~(@p9@$0 <=> @p10@$0) | ~(@p9@$1 <=> @p10@$1));

pred Etage_lesseq(var0 @p11@$0, @p11@$1, var0 @p12@$0, @p12@$1) =
  ~@p11@$1 & @p12@$1 |
  (@p11@$1 <=> @p12@$1) & (~@p11@$0 & @p12@$0 | (@p11@$0 <=> @p12@$0) & false);

pred is_Level(var0 @p17@$set, var0 @p17@$reset, var0 @p17@$req) =
  is_bool(@p17@$set) & is_bool(@p17@$reset) & is_bool(@p17@$req);

pred Level_eq(var0 @p18@$set_1,
              var0 @p18@$reset_1,
              var0 @p18@$req_1,
              var0 @p18@$set_2,
              var0 @p18@$reset_2,
              var0 @p18@$req_2) =
  (@p18@$set_1 <=> @p18@$set_2) &
  (@p18@$reset_1 <=> @p18@$reset_2) & (@p18@$req_1 <=> @p18@$req_2);

pred is_Levels(var0 @p19@$0$set,
               var0 @p19@$0$reset,
               var0 @p19@$0$req,
               var0 @p19@$1$set,
               var0 @p19@$1$reset,
               var0 @p19@$1$req,
               var0 @p19@$2$set,
               var0 @p19@$2$reset,
               var0 @p19@$2$req,
               var0 @p19@$3$set,
               var0 @p19@$3$reset,
               var0 @p19@$3$req) =
  is_Level(@p19@$0$set, @p19@$0$reset, @p19@$0$req) &
  is_Level(@p19@$1$set, @p19@$1$reset, @p19@$1$req) &
  is_Level(@p19@$2$set, @p19@$2$reset, @p19@$2$req) &
  is_Level(@p19@$3$set, @p19@$3$reset, @p19@$3$req);
```

```
pred Levels_eq(var0 @p20@$0$set_1,
               var0 @p20@$0$reset_1,
               var0 @p20@$0$req_1,
               var0 @p20@$1$set_1,
               var0 @p20@$1$reset_1,
               var0 @p20@$1$req_1,
               var0 @p20@$2$set_1,
               var0 @p20@$2$reset_1,
               var0 @p20@$2$req_1,
               var0 @p20@$3$set_1,
               var0 @p20@$3$reset_1,
               var0 @p20@$3$req_1,
               var0 @p20@$0$set_2,
               var0 @p20@$0$reset_2,
               var0 @p20@$0$req_2,
               var0 @p20@$1$set_2,
               var0 @p20@$1$reset_2,
               var0 @p20@$1$req_2,
               var0 @p20@$2$set_2,
               var0 @p20@$2$reset_2,
               var0 @p20@$2$req_2,
               var0 @p20@$3$set_2,
               var0 @p20@$3$reset_2,
               var0 @p20@$3$req_2) =
  (@p20@$0$set_1 <=> @p20@$0$set_2) &
  (@p20@$0$reset_1 <=> @p20@$0$reset_2) &
  (@p20@$0$req_1 <=> @p20@$0$req_2) &
  (@p20@$1$set_1 <=> @p20@$1$set_2) &
  (@p20@$1$reset_1 <=> @p20@$1$reset_2) &
  (@p20@$1$req_1 <=> @p20@$1$req_2) &
  (@p20@$2$set_1 <=> @p20@$2$set_2) &
  (@p20@$2$reset_1 <=> @p20@$2$reset_2) &
  (@p20@$2$req_1 <=> @p20@$2$req_2) &
  (@p20@$3$set_1 <=> @p20@$3$set_2) &
  (@p20@$3$reset_1 <=> @p20@$3$reset_2) & (@p20@$3$req_1 <=> @p20@$3$req_2);

pred is_Goal(var0 @p21@0, @p21@1, @p21@2) =
  ~(@p21@0 & ~@p21@1 & @p21@2 |
    ~@p21@0 & @p21@1 & @p21@2 | @p21@0 & @p21@1 & @p21@2);

pred Goal_eq(var0 @p22@$0_1, @p22@$1_1, @p22@$2_1,
             var0 @p22@$0_2, @p22@$1_2, @p22@$2_2) =
  (@p22@$0_1 <=> @p22@$0_2) &
  (@p22@$1_1 <=> @p22@$1_2) & (@p22@$2_1 <=> @p22@$2_2);

pred is_Goal_0(var0 @p30@0, @p30@1, @p30@2) = ~@p30@0 & ~@p30@1 & ~@p30@2;
pred is_Goal_1(var0 @p31@0, @p31@1, @p31@2) = @p31@0 & ~@p31@1 & ~@p31@2;
pred is_Goal_2(var0 @p32@0, @p32@1, @p32@2) = ~@p32@0 & @p32@1 & ~@p32@2;
pred is_Goal_3(var0 @p33@0, @p33@1, @p33@2) = @p33@0 & @p33@1 & ~@p33@2;
pred is_Goal_4(var0 @p34@0, @p34@1, @p34@2) = ~@p34@0 & ~@p34@1 & @p34@2;
pred Goal_add(var0 @p23@$0, @p23@$1, @p23@$2,
              var0 @p24@$0, @p24@$1, @p24@$2,
              var0 @p25@$0, @p25@$1, @p25@$2) =
  (@p25@$0 <=> ~(@p23@$0 <=> ~(@p24@$0 <=> false))) &
  (@p25@$1 <=>
   ~(@p23@$1 <=>
     ~(@p24@$1 <=> (false & ~(@p23@$0 <=> @p24@$0) | @p23@$0 & @p24@$0)))) &
  (@p25@$2 <=>
   ~(@p23@$2 <=>
     ~(@p24@$2 <=>
       ((false & ~(@p23@$0 <=> @p24@$0) | @p23@$0 & @p24@$0) &
        ~(@p23@$1 <=> @p24@$1) | @p23@$1 & @p24@$1)))) &
  ~(((false & ~(@p23@$0 <=> @p24@$0) | @p23@$0 & @p24@$0) &
     ~(@p23@$1 <=> @p24@$1) | @p23@$1 & @p24@$1) & ~(@p23@$2 <=> @p24@$2) |
    @p23@$2 & @p24@$2);

pred Goal_less(var0 @p26@$0, @p26@$1, @p26@$2, var0 @p27@$0, @p27@$1, @p27@$2) =
  (~@p26@$2 & @p27@$2 |
   (@p26@$2 <=> @p27@$2) &
   (~@p26@$1 & @p27@$1 |
    (@p26@$1 <=> @p27@$1) &
    (~@p26@$0 & @p27@$0 | (@p26@$0 <=> @p27@$0) & false))) &
  (~(@p26@$0 <=> @p27@$0) | ~(@p26@$1 <=> @p27@$1) | ~(@p26@$2 <=> @p27@$2));

pred Goal_lesseq(var0 @p28@$0, @p28@$1, @p28@$2, var0 @p29@$0, @p29@$1, @p29@$2)
  =
  ~@p28@$2 & @p29@$2 |
  (@p28@$2 <=> @p29@$2) &
  (~@p28@$1 & @p29@$1 |
   (@p28@$1 <=> @p29@$1) &
   (~@p28@$0 & @p29@$0 | (@p28@$0 <=> @p29@$0) & false));

pred is_Dir(var0 @p35@0, @p35@1) = ~(@p35@0 & @p35@1);
pred Dir_eq(var0 @p36@$0_1, @p36@$1_1, var0 @p36@$0_2, @p36@$1_2) =
  (@p36@$0_1 <=> @p36@$0_2) & (@p36@$1_1 <=> @p36@$1_2);

pred is_Dir_up(var0 @p37@0, @p37@1) = ~@p37@0 & ~@p37@1;
pred is_Dir_down(var0 @p38@0, @p38@1) = @p38@0 & ~@p38@1;
pred is_Dir_none(var0 @p39@0, @p39@1) = ~@p39@0 & @p39@1;
pred is_Door(var0 @p40@0, @p40@1) = true;
pred Door_eq(var0 @p41@$0_1, @p41@$1_1, var0 @p41@$0_2, @p41@$1_2) =
  (@p41@$0_1 <=> @p41@$0_2) & (@p41@$1_1 <=> @p41@$1_2);

pred is_Door_open(var0 @p42@0, @p42@1) = ~@p42@0 & ~@p42@1;
pred is_Door_opening(var0 @p43@0, @p43@1) = @p43@0 & ~@p43@1;
pred is_Door_closed(var0 @p44@0, @p44@1) = ~@p44@0 & @p44@1;
pred is_Door_closing(var0 @p45@0, @p45@1) = @p45@0 & @p45@1;
pred is_Ppd(var0 @p46@0, @p46@1) = ~(@p46@0 & @p46@1);
pred Ppd_eq(var0 @p47@$0_1, @p47@$1_1, var0 @p47@$0_2, @p47@$1_2) =
  (@p47@$0_1 <=> @p47@$0_2) & (@p47@$1_1 <=> @p47@$1_2);

pred is_Ppd_nobody(var0 @p48@0, @p48@1) = ~@p48@0 & ~@p48@1;
pred is_Ppd_somebody(var0 @p49@0, @p49@1) = @p49@0 & ~@p49@1;
pred is_Ppd_vip(var0 @p50@0, @p50@1) = ~@p50@0 & @p50@1;
pred is_State(var0 @p51@$levels$0$set,
              var0 @p51@$levels$0$reset,
              var0 @p51@$levels$0$req,
              var0 @p51@$levels$1$set,
              var0 @p51@$levels$1$reset,
              var0 @p51@$levels$1$req,
              var0 @p51@$levels$2$set,
              var0 @p51@$levels$2$reset,
              var0 @p51@$levels$2$req,
              var0 @p51@$levels$3$set,
              var0 @p51@$levels$3$reset,
              var0 @p51@$levels$3$req,
              var0 @p51@$pos$0, @p51@$pos$1,
              var0 @p51@$goal$0, @p51@$goal$1, @p51@$goal$2,
              var0 @p51@$dir$0, @p51@$dir$1,
              var0 @p51@$door$0, @p51@$door$1,
              var0 @p51@$ppd$0, @p51@$ppd$1) =
  is_Levels(@p51@$levels$0$set, @p51@$levels$0$reset, @p51@$levels$0$req,
            @p51@$levels$1$set, @p51@$levels$1$reset, @p51@$levels$1$req,
            @p51@$levels$2$set, @p51@$levels$2$reset, @p51@$levels$2$req,
            @p51@$levels$3$set, @p51@$levels$3$reset, @p51@$levels$3$req) &
  is_Etage(@p51@$pos$0, @p51@$pos$1) &
  is_Goal(@p51@$goal$0, @p51@$goal$1, @p51@$goal$2) &
  is_Dir(@p51@$dir$0, @p51@$dir$1) &
  is_Door(@p51@$door$0, @p51@$door$1) & is_Ppd(@p51@$ppd$0, @p51@$ppd$1);

pred State_eq(var0 @p52@$levels$0$set_1,
              var0 @p52@$levels$0$reset_1,
              var0 @p52@$levels$0$req_1,
              var0 @p52@$levels$1$set_1,
              var0 @p52@$levels$1$reset_1,
              var0 @p52@$levels$1$req_1,
              var0 @p52@$levels$2$set_1,
              var0 @p52@$levels$2$reset_1,
              var0 @p52@$levels$2$req_1,
              var0 @p52@$levels$3$set_1,
              var0 @p52@$levels$3$reset_1,
              var0 @p52@$levels$3$req_1,
              var0 @p52@$pos$0_1, @p52@$pos$1_1,
              var0 @p52@$goal$0_1, @p52@$goal$1_1, @p52@$goal$2_1,
              var0 @p52@$dir$0_1, @p52@$dir$1_1,
              var0 @p52@$door$0_1, @p52@$door$1_1,
              var0 @p52@$ppd$0_1, @p52@$ppd$1_1,
              var0 @p52@$levels$0$set_2,
              var0 @p52@$levels$0$reset_2,
              var0 @p52@$levels$0$req_2,
              var0 @p52@$levels$1$set_2,
              var0 @p52@$levels$1$reset_2,
              var0 @p52@$levels$1$req_2,
              var0 @p52@$levels$2$set_2,
              var0 @p52@$levels$2$reset_2,
              var0 @p52@$levels$2$req_2,
              var0 @p52@$levels$3$set_2,
              var0 @p52@$levels$3$reset_2,
              var0 @p52@$levels$3$req_2,
              var0 @p52@$pos$0_2, @p52@$pos$1_2,
              var0 @p52@$goal$0_2, @p52@$goal$1_2, @p52@$goal$2_2,
              var0 @p52@$dir$0_2, @p52@$dir$1_2,
              var0 @p52@$door$0_2, @p52@$door$1_2,
              var0 @p52@$ppd$0_2, @p52@$ppd$1_2) =
  (@p52@$levels$0$set_1 <=> @p52@$levels$0$set_2) &
  (@p52@$levels$0$reset_1 <=> @p52@$levels$0$reset_2) &
  (@p52@$levels$0$req_1 <=> @p52@$levels$0$req_2) &
  (@p52@$levels$1$set_1 <=> @p52@$levels$1$set_2) &
  (@p52@$levels$1$reset_1 <=> @p52@$levels$1$reset_2) &
  (@p52@$levels$1$req_1 <=> @p52@$levels$1$req_2) &
  (@p52@$levels$2$set_1 <=> @p52@$levels$2$set_2) &
  (@p52@$levels$2$reset_1 <=> @p52@$levels$2$reset_2) &
  (@p52@$levels$2$req_1 <=> @p52@$levels$2$req_2) &
  (@p52@$levels$3$set_1 <=> @p52@$levels$3$set_2) &
  (@p52@$levels$3$reset_1 <=> @p52@$levels$3$reset_2) &
  (@p52@$levels$3$req_1 <=> @p52@$levels$3$req_2) &
  (@p52@$pos$0_1 <=> @p52@$pos$0_2) & (@p52@$pos$1_1 <=> @p52@$pos$1_2) &
  (@p52@$goal$0_1 <=> @p52@$goal$0_2) &
  (@p52@$goal$1_1 <=> @p52@$goal$1_2) & (@p52@$goal$2_1 <=> @p52@$goal$2_2) &
  (@p52@$dir$0_1 <=> @p52@$dir$0_2) & (@p52@$dir$1_1 <=> @p52@$dir$1_2) &
  (@p52@$door$0_1 <=> @p52@$door$0_2) & (@p52@$door$1_1 <=> @p52@$door$1_2) &
  (@p52@$ppd$0_1 <=> @p52@$ppd$0_2) & (@p52@$ppd$1_1 <=> @p52@$ppd$1_2);

pred is_Lift(var2 @p53@$levels$0$set,
             var2 @p53@$levels$0$reset,
             var2 @p53@$levels$0$req,
             var2 @p53@$levels$1$set,
             var2 @p53@$levels$1$reset,
             var2 @p53@$levels$1$req,
             var2 @p53@$levels$2$set,
             var2 @p53@$levels$2$reset,
             var2 @p53@$levels$2$req,
             var2 @p53@$levels$3$set,
             var2 @p53@$levels$3$reset,
             var2 @p53@$levels$3$req,
             var2 @p53@$pos$0, @p53@$pos$1,
             var2 @p53@$goal$0, @p53@$goal$1, @p53@$goal$2,
             var2 @p53@$dir$0, @p53@$dir$1,
             var2 @p53@$door$0, @p53@$door$1,
             var2 @p53@$ppd$0, @p53@$ppd$1) =
  all1 @p54@:
    is_State(@p54@ in @p53@$levels$0$set, @p54@ in @p53@$levels$0$reset,
             @p54@ in @p53@$levels$0$req, @p54@ in @p53@$levels$1$set,
             @p54@ in @p53@$levels$1$reset, @p54@ in @p53@$levels$1$req,
             @p54@ in @p53@$levels$2$set, @p54@ in @p53@$levels$2$reset,
             @p54@ in @p53@$levels$2$req, @p54@ in @p53@$levels$3$set,
             @p54@ in @p53@$levels$3$reset, @p54@ in @p53@$levels$3$req,
             @p54@ in @p53@$pos$0, @p54@ in @p53@$pos$1, @p54@ in @p53@$goal$0,
             @p54@ in @p53@$goal$1, @p54@ in @p53@$goal$2, @p54@ in @p53@$dir$0,
             @p54@ in @p53@$dir$1, @p54@ in @p53@$door$0, @p54@ in @p53@$door$1,
             @p54@ in @p53@$ppd$0, @p54@ in @p53@$ppd$1);

pred Lift_eq(var2 @p55@$levels$0$set_1,
             var2 @p55@$levels$0$reset_1,
             var2 @p55@$levels$0$req_1,
             var2 @p55@$levels$1$set_1,
             var2 @p55@$levels$1$reset_1,
             var2 @p55@$levels$1$req_1,
             var2 @p55@$levels$2$set_1,
             var2 @p55@$levels$2$reset_1,
             var2 @p55@$levels$2$req_1,
             var2 @p55@$levels$3$set_1,
             var2 @p55@$levels$3$reset_1,
             var2 @p55@$levels$3$req_1,
             var2 @p55@$pos$0_1, @p55@$pos$1_1,
             var2 @p55@$goal$0_1, @p55@$goal$1_1, @p55@$goal$2_1,
             var2 @p55@$dir$0_1, @p55@$dir$1_1,
             var2 @p55@$door$0_1, @p55@$door$1_1,
             var2 @p55@$ppd$0_1, @p55@$ppd$1_1,
             var2 @p55@$levels$0$set_2,
             var2 @p55@$levels$0$reset_2,
             var2 @p55@$levels$0$req_2,
             var2 @p55@$levels$1$set_2,
             var2 @p55@$levels$1$reset_2,
             var2 @p55@$levels$1$req_2,
             var2 @p55@$levels$2$set_2,
             var2 @p55@$levels$2$reset_2,
             var2 @p55@$levels$2$req_2,
             var2 @p55@$levels$3$set_2,
             var2 @p55@$levels$3$reset_2,
             var2 @p55@$levels$3$req_2,
             var2 @p55@$pos$0_2, @p55@$pos$1_2,
             var2 @p55@$goal$0_2, @p55@$goal$1_2, @p55@$goal$2_2,
             var2 @p55@$dir$0_2, @p55@$dir$1_2,
             var2 @p55@$door$0_2, @p55@$door$1_2,
             var2 @p55@$ppd$0_2, @p55@$ppd$1_2) =
  (@p55@$levels$0$set_1 = @p55@$levels$0$set_2) &
  (@p55@$levels$0$reset_1 = @p55@$levels$0$reset_2) &
  (@p55@$levels$0$req_1 = @p55@$levels$0$req_2) &
  (@p55@$levels$1$set_1 = @p55@$levels$1$set_2) &
  (@p55@$levels$1$reset_1 = @p55@$levels$1$reset_2) &
  (@p55@$levels$1$req_1 = @p55@$levels$1$req_2) &
  (@p55@$levels$2$set_1 = @p55@$levels$2$set_2) &
  (@p55@$levels$2$reset_1 = @p55@$levels$2$reset_2) &
  (@p55@$levels$2$req_1 = @p55@$levels$2$req_2) &
  (@p55@$levels$3$set_1 = @p55@$levels$3$set_2) &
  (@p55@$levels$3$reset_1 = @p55@$levels$3$reset_2) &
  (@p55@$levels$3$req_1 = @p55@$levels$3$req_2) &
  (@p55@$pos$0_1 = @p55@$pos$0_2) & (@p55@$pos$1_1 = @p55@$pos$1_2) &
  (@p55@$goal$0_1 = @p55@$goal$0_2) &
  (@p55@$goal$1_1 = @p55@$goal$1_2) & (@p55@$goal$2_1 = @p55@$goal$2_2) &
  (@p55@$dir$0_1 = @p55@$dir$0_2) & (@p55@$dir$1_1 = @p55@$dir$1_2) &
  (@p55@$door$0_1 = @p55@$door$0_2) & (@p55@$door$1_1 = @p55@$door$1_2) &
  (@p55@$ppd$0_1 = @p55@$ppd$0_2) & (@p55@$ppd$1_1 = @p55@$ppd$1_2);

pred init_level(var0 l$set, var0 l$reset, var0 l$req) = l$req;
pred init(var0 s$levels$0$set,
          var0 s$levels$0$reset,
          var0 s$levels$0$req,
          var0 s$levels$1$set,
          var0 s$levels$1$reset,
          var0 s$levels$1$req,
          var0 s$levels$2$set,
          var0 s$levels$2$reset,
          var0 s$levels$2$req,
          var0 s$levels$3$set,
          var0 s$levels$3$reset,
          var0 s$levels$3$req,
          var0 s$pos$0, s$pos$1,
          var0 s$goal$0, s$goal$1, s$goal$2,
          var0 s$dir$0, s$dir$1,
          var0 s$door$0, s$door$1,
          var0 s$ppd$0, s$ppd$1) =
  init_level(s$levels$0$set, s$levels$0$reset, s$levels$0$req) &
  init_level(s$levels$1$set, s$levels$1$reset, s$levels$1$req) &
  init_level(s$levels$2$set, s$levels$2$reset, s$levels$2$req) &
  init_level(s$levels$3$set, s$levels$3$reset, s$levels$3$req) &
  is_Etage_1(s$pos$0, s$pos$1) & is_Goal_4(s$goal$0, s$goal$1, s$goal$2) &
  is_Dir_none(s$dir$0, s$dir$1) & is_Door_open(s$door$0, s$door$1) &
  is_Ppd_nobody(s$ppd$0, s$ppd$1);

pred stop(var0 s$levels$0$set,
          var0 s$levels$0$reset,
          var0 s$levels$0$req,
          var0 s$levels$1$set,
          var0 s$levels$1$reset,
          var0 s$levels$1$req,
          var0 s$levels$2$set,
          var0 s$levels$2$reset,
          var0 s$levels$2$req,
          var0 s$levels$3$set,
          var0 s$levels$3$reset,
          var0 s$levels$3$req,
          var0 s$pos$0, s$pos$1,
          var0 s$goal$0, s$goal$1, s$goal$2,
          var0 s$dir$0, s$dir$1,
          var0 s$door$0, s$door$1,
          var0 s$ppd$0, s$ppd$1) =
  Etage_eq(s$pos$0, s$pos$1, false, false) & s$levels$0$req |
  Etage_eq(s$pos$0, s$pos$1, true, false) & s$levels$1$req |
  Etage_eq(s$pos$0, s$pos$1, false, true) & s$levels$2$req |
  Etage_eq(s$pos$0, s$pos$1, true, true) & s$levels$3$req |
  Goal_eq(s$goal$0, s$goal$1, s$goal$2, s$pos$0, s$pos$1, false) |
  is_Etage_0(s$pos$0, s$pos$1) & is_Dir_down(s$dir$0, s$dir$1) |
  is_Etage_3(s$pos$0, s$pos$1) & is_Dir_up(s$dir$0, s$dir$1);

pred extern_up_req(var0 s$levels$0$set,
                   var0 s$levels$0$reset,
                   var0 s$levels$0$req,
                   var0 s$levels$1$set,
                   var0 s$levels$1$reset,
                   var0 s$levels$1$req,
                   var0 s$levels$2$set,
                    var0 s$levels$2$reset,
                    var0 s$levels$2$req,
                    var0 s$levels$3$set,
                    var0 s$levels$3$reset,
                    var0 s$levels$3$req,
                    var0 s$pos$0, s$pos$1,
                    var0 s$goal$0, s$goal$1, s$goal$2,
                    var0 s$dir$0, s$dir$1,
                    var0 s$door$0, s$door$1,
                    var0 s$ppd$0, s$ppd$1) =
  ~true & s$levels$0$req & Etage_less(s$pos$0, s$pos$1, false, false) |
  ~false & s$levels$1$req & Etage_less(s$pos$0, s$pos$1, true, false) |
  ~false & s$levels$2$req & Etage_less(s$pos$0, s$pos$1, false, true) |
  ~false & s$levels$3$req & Etage_less(s$pos$0, s$pos$1, true, true);

pred extern_down_req(var0 s$levels$0$set,
                     var0 s$levels$0$reset,
                     var0 s$levels$0$req,
                     var0 s$levels$1$set,
                     var0 s$levels$1$reset,
                     var0 s$levels$1$req,
                     var0 s$levels$2$set,
                     var0 s$levels$2$reset,
                     var0 s$levels$2$req,
                     var0 s$levels$3$set,
                     var0 s$levels$3$reset,
                     var0 s$levels$3$req,
                     var0 s$pos$0, s$pos$1,
                     var0 s$goal$0, s$goal$1, s$goal$2,
                     var0 s$dir$0, s$dir$1,
                     var0 s$door$0, s$door$1,
                     var0 s$ppd$0, s$ppd$1) =
  ~false & s$levels$0$req & Etage_less(false, false, s$pos$0, s$pos$1) |
  ~false & s$levels$1$req & Etage_less(true, false, s$pos$0, s$pos$1) |
  ~false & s$levels$2$req & Etage_less(false, true, s$pos$0, s$pos$1) |
  ~true & s$levels$3$req & Etage_less(true, true, s$pos$0, s$pos$1);

pred go_up(var0 s$levels$0$set,
           var0 s$levels$0$reset,
           var0 s$levels$0$req,
           var0 s$levels$1$set,
           var0 s$levels$1$reset,
           var0 s$levels$1$req,
           var0 s$levels$2$set,
           var0 s$levels$2$reset,
           var0 s$levels$2$req,
           var0 s$levels$3$set,
           var0 s$levels$3$reset,
           var0 s$levels$3$req,
           var0 s$pos$0, s$pos$1,
           var0 s$goal$0, s$goal$1, s$goal$2,
           var0 s$dir$0, s$dir$1,
           var0 s$door$0, s$door$1,
           var0 s$ppd$0, s$ppd$1) =
  ~is_Goal_4(s$goal$0, s$goal$1, s$goal$2) &
  Goal_less(s$pos$0, s$pos$1, false, s$goal$0, s$goal$1, s$goal$2);

pred go_down(var0 s$levels$0$set,
             var0 s$levels$0$reset,
             var0 s$levels$0$req,
             var0 s$levels$1$set,
             var0 s$levels$1$reset,
             var0 s$levels$1$req,
             var0 s$levels$2$set,
             var0 s$levels$2$reset,
             var0 s$levels$2$req,
             var0 s$levels$3$set,
             var0 s$levels$3$reset,
             var0 s$levels$3$req,
             var0 s$pos$0, s$pos$1,
             var0 s$goal$0, s$goal$1, s$goal$2,
             var0 s$dir$0, s$dir$1,
             var0 s$door$0, s$door$1,
             var0 s$ppd$0, s$ppd$1) =
  ~is_Goal_4(s$goal$0, s$goal$1, s$goal$2) &
  Goal_less(s$goal$0, s$goal$1, s$goal$2, s$pos$0, s$pos$1, false);

pred next_door(var0 s$levels$0$set,
               var0 s$levels$0$reset,
               var0 s$levels$0$req,
               var0 s$levels$1$set,
               var0 s$levels$1$reset,
               var0 s$levels$1$req,
               var0 s$levels$2$set,
               var0 s$levels$2$reset,
               var0 s$levels$2$req,
               var0 s$levels$3$set,
               var0 s$levels$3$reset,
               var0 s$levels$3$req,
               var0 s$pos$0, s$pos$1,
               var0 s$goal$0, s$goal$1, s$goal$2,
               var0 s$dir$0, s$dir$1,
               var0 s$door$0, s$door$1,
               var0 s$ppd$0, s$ppd$1,
               var0 t$levels$0$set,
               var0 t$levels$0$reset,
               var0 t$levels$0$req,
               var0 t$levels$1$set,
               var0 t$levels$1$reset,
               var0 t$levels$1$req,
               var0 t$levels$2$set,
               var0 t$levels$2$reset,
               var0 t$levels$2$req,
               var0 t$levels$3$set,
               var0 t$levels$3$reset,
               var0 t$levels$3$req,
               var0 t$pos$0, t$pos$1,
               var0 t$goal$0, t$goal$1, t$goal$2,
               var0 t$dir$0, t$dir$1,
               var0 t$door$0, t$door$1,
               var0 t$ppd$0, t$ppd$1) =
  (stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
        s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
        s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
        s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
        s$door$0, s$door$1, s$ppd$0, s$ppd$1) =>
   (is_Door_closed(s$door$0, s$door$1) => is_Door_opening(t$door$0, t$door$1)) &
   (is_Door_opening(s$door$0, s$door$1) => is_Door_open(t$door$0, t$door$1)) &
   (is_Door_open(s$door$0, s$door$1) => is_Door_open(t$door$0, t$door$1)) &
   (is_Door_closing(s$door$0, s$door$1) => is_Door_closing(t$door$0, t$door$1)))
  &
  (~stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
         s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
         s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
         s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
         s$door$0, s$door$1, s$ppd$0, s$ppd$1) =>
   (is_Door_open(s$door$0, s$door$1) => is_Door_closing(t$door$0, t$door$1)) &
   (is_Door_closing(s$door$0, s$door$1) => is_Door_closed(t$door$0, t$door$1)) &
   (is_Door_opening(s$door$0, s$door$1) => is_Door_opening(t$door$0, t$door$1))
   &
   (is_Door_closed(s$door$0, s$door$1) => is_Door_closed(t$door$0, t$door$1)));

pred next_pos(var0 s$levels$0$set,
              var0 s$levels$0$reset,
              var0 s$levels$0$req,
              var0 s$levels$1$set,
              var0 s$levels$1$reset,
              var0 s$levels$1$req,
              var0 s$levels$2$set,
              var0 s$levels$2$reset,
              var0 s$levels$2$req,
              var0 s$levels$3$set,
              var0 s$levels$3$reset,
              var0 s$levels$3$req,
              var0 s$pos$0, s$pos$1,
              var0 s$goal$0, s$goal$1, s$goal$2,
              var0 s$dir$0, s$dir$1,
              var0 s$door$0, s$door$1,
              var0 s$ppd$0, s$ppd$1,
              var0 t$levels$0$set,
              var0 t$levels$0$reset,
              var0 t$levels$0$req,
              var0 t$levels$1$set,
              var0 t$levels$1$reset,
              var0 t$levels$1$req,
              var0 t$levels$2$set,
              var0 t$levels$2$reset,
              var0 t$levels$2$req,
              var0 t$levels$3$set,
              var0 t$levels$3$reset,
              var0 t$levels$3$req,
              var0 t$pos$0, t$pos$1,
              var0 t$goal$0, t$goal$1, t$goal$2,
              var0 t$dir$0, t$dir$1,
              var0 t$door$0, t$door$1,
              var0 t$ppd$0, t$ppd$1) =
  (stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
        s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
        s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
        s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
        s$door$0, s$door$1, s$ppd$0, s$ppd$1) =>
   (is_Dir_up(s$dir$0, s$dir$1) =>
    Etage_add(s$pos$0, s$pos$1, true, false, t$pos$0, t$pos$1)) &
   (is_Dir_down(s$dir$0, s$dir$1) =>
    Etage_add(t$pos$0, t$pos$1, true, false, s$pos$0, s$pos$1)) &
   (is_Dir_none(s$dir$0, s$dir$1) =>
    Etage_eq(t$pos$0, t$pos$1, s$pos$0, s$pos$1))) &
  (~stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
         s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
         s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
         s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
         s$door$0, s$door$1, s$ppd$0, s$ppd$1) =>
   Etage_eq(t$pos$0, t$pos$1, s$pos$0, s$pos$1));

pred next_ppd(var0 s$levels$0$set,
              var0 s$levels$0$reset,
              var0 s$levels$0$req,
              var0 s$levels$1$set,
              var0 s$levels$1$reset,
              var0 s$levels$1$req,
              var0 s$levels$2$set,
              var0 s$levels$2$reset,
              var0 s$levels$2$req,
              var0 s$levels$3$set,
              var0 s$levels$3$reset,
              var0 s$levels$3$req,
              var0 s$pos$0, s$pos$1,
              var0 s$goal$0, s$goal$1, s$goal$2,
              var0 s$dir$0, s$dir$1,
              var0 s$door$0, s$door$1,
              var0 s$ppd$0, s$ppd$1,
              var0 t$levels$0$set,
              var0 t$levels$0$reset,
              var0 t$levels$0$req,
              var0 t$levels$1$set,
              var0 t$levels$1$reset,
              var0 t$levels$1$req,
              var0 t$levels$2$set,
              var0 t$levels$2$reset,
              var0 t$levels$2$req,
              var0 t$levels$3$set,
              var0 t$levels$3$reset,
              var0 t$levels$3$req,
              var0 t$pos$0, t$pos$1,
              var0 t$goal$0, t$goal$1, t$goal$2,
              var0 t$dir$0, t$dir$1,
              var0 t$door$0, t$door$1,
              var0 t$ppd$0, t$ppd$1) =
  is_Door_open(s$door$0, s$door$1) &
  (is_Etage_0(s$pos$0, s$pos$1) =>
   (is_Ppd_nobody(t$ppd$0, t$ppd$1) | is_Ppd_vip(t$ppd$0, t$ppd$1))) |
  ~is_Door_open(s$door$0, s$door$1) &
  true & Ppd_eq(t$ppd$0, t$ppd$1, s$ppd$0, s$ppd$1);

pred next_goal(var0 s$levels$0$set,
               var0 s$levels$0$reset,
               var0 s$levels$0$req,
               var0 s$levels$1$set,
               var0 s$levels$1$reset,
               var0 s$levels$1$req,
               var0 s$levels$2$set,
               var0 s$levels$2$reset,
               var0 s$levels$2$req,
               var0 s$levels$3$set,
               var0 s$levels$3$reset,
               var0 s$levels$3$req,
               var0 s$pos$0, s$pos$1,
               var0 s$goal$0, s$goal$1, s$goal$2,
               var0 s$dir$0, s$dir$1,
               var0 s$door$0, s$door$1,
               var0 s$ppd$0, s$ppd$1,
               var0 t$levels$0$set,
               var0 t$levels$0$reset,
               var0 t$levels$0$req,
               var0 t$levels$1$set,
               var0 t$levels$1$reset,
               var0 t$levels$1$req,
               var0 t$levels$2$set,
               var0 t$levels$2$reset,
               var0 t$levels$2$req,
               var0 t$levels$3$set,
               var0 t$levels$3$reset,
               var0 t$levels$3$req,
               var0 t$pos$0, t$pos$1,
               var0 t$goal$0, t$goal$1, t$goal$2,
               var0 t$dir$0, t$dir$1,
               var0 t$door$0, t$door$1,
               var0 t$ppd$0, t$ppd$1) =
  Goal_eq(s$pos$0, s$pos$1, false, s$goal$0, s$goal$1, s$goal$2) &
  is_Door_opening(s$door$0, s$door$1) & is_Goal_4(t$goal$0, t$goal$1, t$goal$2)
  |
  ~(Goal_eq(s$pos$0, s$pos$1, false, s$goal$0, s$goal$1, s$goal$2) &
    is_Door_opening(s$door$0, s$door$1)) &
  is_Door_open(s$door$0, s$door$1) & ~is_Ppd_nobody(s$ppd$0, s$ppd$1) &
  (is_Ppd_somebody(s$ppd$0, s$ppd$1) =>
   ~is_Goal_0(t$goal$0, t$goal$1, t$goal$2) &
   ~is_Goal_4(t$goal$0, t$goal$1, t$goal$2)) &
  (is_Ppd_vip(s$ppd$0, s$ppd$1) => ~is_Goal_4(t$goal$0, t$goal$1, t$goal$2)) |
  ~(is_Door_open(s$door$0, s$door$1) & ~is_Ppd_nobody(s$ppd$0, s$ppd$1)) &
  ~(Goal_eq(s$pos$0, s$pos$1, false, s$goal$0, s$goal$1, s$goal$2) &
    is_Door_opening(s$door$0, s$door$1)) &
  true & Goal_eq(t$goal$0, t$goal$1, t$goal$2, s$goal$0, s$goal$1, s$goal$2);

pred right_dir(var0 s$levels$0$set,
               var0 s$levels$0$reset,
               var0 s$levels$0$req,
               var0 s$levels$1$set,
               var0 s$levels$1$reset,
               var0 s$levels$1$req,
               var0 s$levels$2$set,
               var0 s$levels$2$reset,
               var0 s$levels$2$req,
               var0 s$levels$3$set,
               var0 s$levels$3$reset,
               var0 s$levels$3$req,
               var0 s$pos$0, s$pos$1,
               var0 s$goal$0, s$goal$1, s$goal$2,
               var0 s$dir$0, s$dir$1,
               var0 s$door$0, s$door$1,
               var0 s$ppd$0, s$ppd$1) =
  (extern_up_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
                 s$levels$1$set, s$levels$1$reset, s$levels$1$req,
                 s$levels$2$set, s$levels$2$reset, s$levels$2$req,
                 s$levels$3$set, s$levels$3$reset, s$levels$3$req, s$pos$0,
                 s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
                 s$door$0, s$door$1, s$ppd$0, s$ppd$1) |
   go_up(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
         s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
         s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
         s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
         s$door$0, s$door$1, s$ppd$0, s$ppd$1)) & is_Dir_up(s$dir$0, s$dir$1) |
  (extern_down_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
                   s$levels$1$set, s$levels$1$reset, s$levels$1$req,
                   s$levels$2$set, s$levels$2$reset, s$levels$2$req,
                   s$levels$3$set, s$levels$3$reset, s$levels$3$req, s$pos$0,
                   s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
                   s$door$0, s$door$1, s$ppd$0, s$ppd$1) |
   go_down(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
           s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
           s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
           s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
           s$door$0, s$door$1, s$ppd$0, s$ppd$1)) &
  is_Dir_down(s$dir$0, s$dir$1);

pred next_dir(var0 s$levels$0$set,
              var0 s$levels$0$reset,
              var0 s$levels$0$req,
              var0 s$levels$1$set,
              var0 s$levels$1$reset,
              var0 s$levels$1$req,
              var0 s$levels$2$set,
              var0 s$levels$2$reset,
              var0 s$levels$2$req,
              var0 s$levels$3$set,
            var0 s$levels$3$reset,
            var0 s$levels$3$req,
            var0 s$pos$0, s$pos$1,
            var0 s$goal$0, s$goal$1, s$goal$2,
            var0 s$dir$0, s$dir$1,
            var0 s$door$0, s$door$1,
            var0 s$ppd$0, s$ppd$1,
            var0 t$levels$0$set,
            var0 t$levels$0$reset,
            var0 t$levels$0$req,
            var0 t$levels$1$set,
            var0 t$levels$1$reset,
            var0 t$levels$1$req,
            var0 t$levels$2$set,
            var0 t$levels$2$reset,
            var0 t$levels$2$req,
            var0 t$levels$3$set,
            var0 t$levels$3$reset,
            var0 t$levels$3$req,
            var0 t$pos$0, t$pos$1,
            var0 t$goal$0, t$goal$1, t$goal$2,
            var0 t$dir$0, t$dir$1,
            var0 t$door$0, t$door$1,
            var0 t$ppd$0, t$ppd$1) =
(stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
      s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
      s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
      s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
      s$door$0, s$door$1, s$ppd$0, s$ppd$1) |
 ~is_Door_closed(s$door$0, s$door$1)) & is_Dir_none(t$dir$0, t$dir$1) |
~(stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
       s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
       s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
       s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
       s$door$0, s$door$1, s$ppd$0, s$ppd$1) |
  ~is_Door_closed(s$door$0, s$door$1)) &
right_dir(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
Dir_eq(t$dir$0, t$dir$1, s$dir$0, s$dir$1) |
~right_dir(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
           s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
           s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
           s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
           s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
~(stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
       s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
       s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
       s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
       s$door$0, s$door$1, s$ppd$0, s$ppd$1) |
  ~is_Door_closed(s$door$0, s$door$1)) &
is_Dir_none(s$dir$0, s$dir$1) &
```

```
(go_up(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
       s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
       s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
       s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
       s$door$0, s$door$1, s$ppd$0, s$ppd$1) & is_Dir_up(t$dir$0, t$dir$1) |
 ~go_up(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
        s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
        s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
        s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
        s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
 go_down(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
         s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
         s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
         s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
         s$door$0, s$door$1, s$ppd$0, s$ppd$1) & is_Dir_down(t$dir$0, t$dir$1)
 |
 ~go_down(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
 ~go_up(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
        s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
        s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
        s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
        s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
 extern_up_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
               s$levels$1$set, s$levels$1$reset, s$levels$1$req,
               s$levels$2$set, s$levels$2$reset, s$levels$2$req,
               s$levels$3$set, s$levels$3$reset, s$levels$3$req, s$pos$0,
               s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
               s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
 is_Dir_up(t$dir$0, t$dir$1) |
 ~extern_up_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
                s$levels$1$set, s$levels$1$reset, s$levels$1$req,
                s$levels$2$set, s$levels$2$reset, s$levels$2$req,
                s$levels$3$set, s$levels$3$reset, s$levels$3$req, s$pos$0,
                s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
                s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
 ~go_down(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
 ~go_up(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
        s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
        s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
        s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
        s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
 extern_down_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
                 s$levels$1$set, s$levels$1$reset, s$levels$1$req,
                 s$levels$2$set, s$levels$2$reset, s$levels$2$req,
                 s$levels$3$set, s$levels$3$reset, s$levels$3$req, s$pos$0,
                 s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
                 s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
   is_Dir_down(t$dir$0, t$dir$1) |
   ~extern_down_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
                    s$levels$1$set, s$levels$1$reset, s$levels$1$req,
                    s$levels$2$set, s$levels$2$reset, s$levels$2$req,
                    s$levels$3$set, s$levels$3$reset, s$levels$3$req, s$pos$0,
                    s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
                    s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
   ~extern_up_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
                  s$levels$1$set, s$levels$1$reset, s$levels$1$req,
                  s$levels$2$set, s$levels$2$reset, s$levels$2$req,
                  s$levels$3$set, s$levels$3$reset, s$levels$3$req, s$pos$0,
                  s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
                  s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
   ~go_down(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
            s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
            s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
            s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
            s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
   ~go_up(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
   true & is_Dir_none(t$dir$0, t$dir$1)) |
  ~is_Dir_none(s$dir$0, s$dir$1) &
  ~right_dir(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
             s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
             s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
             s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
             s$door$0, s$door$1, s$ppd$0, s$ppd$1) &
  ~(stop(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
         s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
         s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
         s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
         s$door$0, s$door$1, s$ppd$0, s$ppd$1) |
    ~is_Door_closed(s$door$0, s$door$1)) & true & is_Dir_none(t$dir$0, t$dir$1);

pred next_req_level(var0 a$set,
                    var0 a$reset,
                    var0 a$req,
                    var0 b$set,
                    var0 b$reset,
                    var0 b$req) =
  a$set & b$req |
  ~a$set & a$reset & ~b$req | ~a$reset & ~a$set & true & (a$req <=> b$req);

pred next_req(var0 s$levels$0$set,
              var0 s$levels$0$reset,
              var0 s$levels$0$req,
              var0 s$levels$1$set,
              var0 s$levels$1$reset,
              var0 s$levels$1$req,
              var0 s$levels$2$set,
              var0 s$levels$2$reset,
              var0 s$levels$2$req,
              var0 s$levels$3$set,
              var0 s$levels$3$reset,
              var0 s$levels$3$req,
              var0 s$pos$0, s$pos$1,
              var0 s$goal$0, s$goal$1, s$goal$2,
              var0 s$dir$0, s$dir$1,
              var0 s$door$0, s$door$1,
              var0 s$ppd$0, s$ppd$1,
              var0 t$levels$0$set,
              var0 t$levels$0$reset,
              var0 t$levels$0$req,
              var0 t$levels$1$set,
              var0 t$levels$1$reset,
              var0 t$levels$1$req,
              var0 t$levels$2$set,
              var0 t$levels$2$reset,
              var0 t$levels$2$req,
              var0 t$levels$3$set,
              var0 t$levels$3$reset,
              var0 t$levels$3$req,
              var0 t$pos$0, t$pos$1,
              var0 t$goal$0, t$goal$1, t$goal$2,
              var0 t$dir$0, t$dir$1,
              var0 t$door$0, t$door$1,
              var0 t$ppd$0, t$ppd$1) =
  next_req_level(s$levels$0$set, s$levels$0$reset, s$levels$0$req,
                 t$levels$0$set, t$levels$0$reset, t$levels$0$req) &
  next_req_level(s$levels$1$set, s$levels$1$reset, s$levels$1$req,
                 t$levels$1$set, t$levels$1$reset, t$levels$1$req) &
  next_req_level(s$levels$2$set, s$levels$2$reset, s$levels$2$req,
                 t$levels$2$set, t$levels$2$reset, t$levels$2$req) &
  next_req_level(s$levels$3$set, s$levels$3$reset, s$levels$3$req,
                 t$levels$3$set, t$levels$3$reset, t$levels$3$req);

pred next(var0 s$levels$0$set,
          var0 s$levels$0$reset,
          var0 s$levels$0$req,
          var0 s$levels$1$set,
          var0 s$levels$1$reset,
          var0 s$levels$1$req,
          var0 s$levels$2$set,
          var0 s$levels$2$reset,
          var0 s$levels$2$req,
          var0 s$levels$3$set,
          var0 s$levels$3$reset,
          var0 s$levels$3$req,
          var0 s$pos$0, s$pos$1,
          var0 s$goal$0, s$goal$1, s$goal$2,
          var0 s$dir$0, s$dir$1,
          var0 s$door$0, s$door$1,
          var0 s$ppd$0, s$ppd$1,
          var0 t$levels$0$set,
        var0 t$levels$0$reset,
        var0 t$levels$0$req,
        var0 t$levels$1$set,
        var0 t$levels$1$reset,
        var0 t$levels$1$req,
        var0 t$levels$2$set,
        var0 t$levels$2$reset,
        var0 t$levels$2$req,
        var0 t$levels$3$set,
        var0 t$levels$3$reset,
        var0 t$levels$3$req,
        var0 t$pos$0, t$pos$1,
        var0 t$goal$0, t$goal$1, t$goal$2,
        var0 t$dir$0, t$dir$1,
        var0 t$door$0, t$door$1,
        var0 t$ppd$0, t$ppd$1) =
next_req(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
         s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
         s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
         s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
         s$door$0, s$door$1, s$ppd$0, s$ppd$1, t$levels$0$set,
         t$levels$0$reset, t$levels$0$req, t$levels$1$set, t$levels$1$reset,
         t$levels$1$req, t$levels$2$set, t$levels$2$reset, t$levels$2$req,
         t$levels$3$set, t$levels$3$reset, t$levels$3$req, t$pos$0, t$pos$1,
         t$goal$0, t$goal$1, t$goal$2, t$dir$0, t$dir$1, t$door$0, t$door$1,
         t$ppd$0, t$ppd$1) &
next_door(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1, t$levels$0$set,
          t$levels$0$reset, t$levels$0$req, t$levels$1$set, t$levels$1$reset,
          t$levels$1$req, t$levels$2$set, t$levels$2$reset, t$levels$2$req,
          t$levels$3$set, t$levels$3$reset, t$levels$3$req, t$pos$0, t$pos$1,
          t$goal$0, t$goal$1, t$goal$2, t$dir$0, t$dir$1, t$door$0, t$door$1,
          t$ppd$0, t$ppd$1) &
next_ppd(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
         s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
         s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
         s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
         s$door$0, s$door$1, s$ppd$0, s$ppd$1, t$levels$0$set,
         t$levels$0$reset, t$levels$0$req, t$levels$1$set, t$levels$1$reset,
         t$levels$1$req, t$levels$2$set, t$levels$2$reset, t$levels$2$req,
         t$levels$3$set, t$levels$3$reset, t$levels$3$req, t$pos$0, t$pos$1,
         t$goal$0, t$goal$1, t$goal$2, t$dir$0, t$dir$1, t$door$0, t$door$1,
         t$ppd$0, t$ppd$1) &
next_goal(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1, t$levels$0$set,
          t$levels$0$reset, t$levels$0$req, t$levels$1$set, t$levels$1$reset,
          t$levels$1$req, t$levels$2$set, t$levels$2$reset, t$levels$2$req,
          t$levels$3$set, t$levels$3$reset, t$levels$3$req, t$pos$0, t$pos$1,
           t$goal$0, t$goal$1, t$goal$2, t$dir$0, t$dir$1, t$door$0, t$door$1,
           t$ppd$0, t$ppd$1) &
  next_pos(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1, t$levels$0$set,
          t$levels$0$reset, t$levels$0$req, t$levels$1$set, t$levels$1$reset,
          t$levels$1$req, t$levels$2$set, t$levels$2$reset, t$levels$2$req,
          t$levels$3$set, t$levels$3$reset, t$levels$3$req, t$pos$0, t$pos$1,
          t$goal$0, t$goal$1, t$goal$2, t$dir$0, t$dir$1, t$door$0, t$door$1,
          t$ppd$0, t$ppd$1) &
  next_dir(s$levels$0$set, s$levels$0$reset, s$levels$0$req, s$levels$1$set,
          s$levels$1$reset, s$levels$1$req, s$levels$2$set, s$levels$2$reset,
          s$levels$2$req, s$levels$3$set, s$levels$3$reset, s$levels$3$req,
          s$pos$0, s$pos$1, s$goal$0, s$goal$1, s$goal$2, s$dir$0, s$dir$1,
          s$door$0, s$door$1, s$ppd$0, s$ppd$1, t$levels$0$set,
          t$levels$0$reset, t$levels$0$req, t$levels$1$set, t$levels$1$reset,
          t$levels$1$req, t$levels$2$set, t$levels$2$reset, t$levels$2$req,
          t$levels$3$set, t$levels$3$reset, t$levels$3$req, t$pos$0, t$pos$1,
          t$goal$0, t$goal$1, t$goal$2, t$dir$0, t$dir$1, t$door$0, t$door$1,
          t$ppd$0, t$ppd$1);

pred run(var2 L$levels$0$set,
         var2 L$levels$0$reset,
         var2 L$levels$0$req,
         var2 L$levels$1$set,
         var2 L$levels$1$reset,
         var2 L$levels$1$req,
         var2 L$levels$2$set,
         var2 L$levels$2$reset,
         var2 L$levels$2$req,
         var2 L$levels$3$set,
         var2 L$levels$3$reset,
         var2 L$levels$3$req,
         var2 L$pos$0, L$pos$1,
         var2 L$goal$0, L$goal$1, L$goal$2,
         var2 L$dir$0, L$dir$1,
         var2 L$door$0, L$door$1,
         var2 L$ppd$0, L$ppd$1) =
  init(0 in L$levels$0$set, 0 in L$levels$0$reset, 0 in L$levels$0$req,
      0 in L$levels$1$set, 0 in L$levels$1$reset, 0 in L$levels$1$req,
      0 in L$levels$2$set, 0 in L$levels$2$reset, 0 in L$levels$2$req,
      0 in L$levels$3$set, 0 in L$levels$3$reset, 0 in L$levels$3$req,
      0 in L$pos$0, 0 in L$pos$1, 0 in L$goal$0, 0 in L$goal$1, 0 in L$goal$2,
      0 in L$dir$0, 0 in L$dir$1, 0 in L$door$0, 0 in L$door$1, 0 in L$ppd$0,
      0 in L$ppd$1) &
  (all1 i:
     is_nat(i) =>
     nat_less(0, i) =>
     next((i - 1) in L$levels$0$set, (i - 1) in L$levels$0$reset,
          (i - 1) in L$levels$0$req, (i - 1) in L$levels$1$set,
          (i - 1) in L$levels$1$reset, (i - 1) in L$levels$1$req,
          (i - 1) in L$levels$2$set, (i - 1) in L$levels$2$reset,
          (i - 1) in L$levels$2$req, (i - 1) in L$levels$3$set,
          (i - 1) in L$levels$3$reset, (i - 1) in L$levels$3$req,
          (i - 1) in L$pos$0, (i - 1) in L$pos$1, (i - 1) in L$goal$0,
          (i - 1) in L$goal$1, (i - 1) in L$goal$2, (i - 1) in L$dir$0,
          (i - 1) in L$dir$1, (i - 1) in L$door$0, (i - 1) in L$door$1,
          (i - 1) in L$ppd$0, (i - 1) in L$ppd$1, i in L$levels$0$set,
          i in L$levels$0$reset, i in L$levels$0$req, i in L$levels$1$set,
          i in L$levels$1$reset, i in L$levels$1$req, i in L$levels$2$set,
          i in L$levels$2$reset, i in L$levels$2$req, i in L$levels$3$set,
          i in L$levels$3$reset, i in L$levels$3$req, i in L$pos$0,
          i in L$pos$1, i in L$goal$0, i in L$goal$1, i in L$goal$2,
          i in L$dir$0, i in L$dir$1, i in L$door$0, i in L$door$1,
          i in L$ppd$0, i in L$ppd$1);

pred spec( var2 L$levels$0$set,
           var2 L$levels$0$reset,
           var2 L$levels$0$req,
           var2 L$levels$1$set,
           var2 L$levels$1$reset,
           var2 L$levels$1$req,
           var2 L$levels$2$set,
           var2 L$levels$2$reset,
           var2 L$levels$2$req,
           var2 L$levels$3$set,
           var2 L$levels$3$reset,
           var2 L$levels$3$req,
           var2 L$pos$0, L$pos$1,
           var2 L$goal$0, L$goal$1, L$goal$2,
           var2 L$dir$0, L$dir$1,
           var2 L$door$0, L$door$1,
           var2 L$ppd$0, L$ppd$1) =
  all1 i:
    is_nat(i) =>
    ~is_Dir_none(i in L$dir$0, i in L$dir$1) =>
    is_Door_closed(i in L$door$0, i in L$door$1);

## Main Goal

var2 L$goal$0,  L$ppd$1,
     L$levels$0$set, L$levels$0$reset, L$levels$0$req, L$levels$1$set,
     L$dir$1, L$ppd$0, L$pos$1, L$door$0, L$door$1,L$goal$1, L$goal$2,
     L$dir$0, L$levels$1$reset, L$levels$1$req, L$levels$2$set,
     L$levels$2$reset, L$levels$2$req, L$levels$3$set,
     L$levels$3$reset, L$levels$3$req, L$pos$0;

run(L$levels$0$set, L$levels$0$reset, L$levels$0$req, L$levels$1$set,
    L$levels$1$reset, L$levels$1$req, L$levels$2$set, L$levels$2$reset,
    L$levels$2$req, L$levels$3$set, L$levels$3$reset, L$levels$3$req,
    L$pos$0, L$pos$1, L$goal$0, L$goal$1, L$goal$2, L$dir$0, L$dir$1,
    L$door$0, L$door$1, L$ppd$0, L$ppd$1)
 =>
 spec(L$levels$0$set, L$levels$0$reset, L$levels$0$req, L$levels$1$set,
    L$levels$1$reset, L$levels$1$req, L$levels$2$set, L$levels$2$reset,
    L$levels$2$req, L$levels$3$set, L$levels$3$reset, L$levels$3$req,
    L$pos$0, L$pos$1, L$goal$0, L$goal$1, L$goal$2, L$dir$0, L$dir$1,
    L$door$0, L$door$1, L$ppd$0, L$ppd$1);
```

C.3 Alternating Bit Protocol

The following is the entire M2L-Str formalization of the alternating bit Protocol.

```
m2l-str;

pred if(var0 a,b,c) = (a=>b) & (~a=>c);
pred onBit  (var1 t, var2 X) = t in X;
pred offBit (var1 t, var2 X) = t notin X;
pred keepBit(var1 t ,var2 X) = t in X <=> t+1 in X;
pred alternateBit(var1 t, var2 X) = t in X <=> t+1 notin X;
pred getnewData(var1 t, var2 D1,D2) = true;
pred keepData(var1 t, var2 D1,D2) = keepBit(t,D1) & keepBit(t,D2);
pred keepTag(var1 t, var2 T1,T2) = keepBit(t,T1) & keepBit(t,T2);
pred transdata(var1 t, var2 SD1,SD2,RD1,RD2) =
     (t in SD1 <=> (t+1) in RD1) &
     (t in SD2 <=> (t+1) in RD2) ;
pred one  (var0 a,b) = ~a & ~ b;
pred two  (var0 a,b) = ~a &   b;
pred three(var0 a,b) =  a & ~ b;
pred for  (var0 a,b) =  a &   b;
pred get (var1 t, var2 A, B) =  t in A    & t in B;
pred send(var1 t, var2 A, B) =  t in A    & t notin B;
pred wait(var1 t, var2 A, B) =  t notin A ;
pred recieve (var1 t, var2 A, B) =  t in A    & t in B;
pred deliver (var1 t, var2 A, B) =  t in A    & t notin B;
pred send_ack(var1 t, var2 A, B) =  t notin A ;
pred mt   (var1 t, var2 A, B) =  t in A & t in B;
pred data0(var1 t, var2 A, B) =  t in A & t notin B;
pred data1(var1 t, var2 A, B) =  t notin A & t in B;
pred error(var1 t, var2 A, B) =  t notin A & t notin B;
pred ack0 (var1 t, var2 A, B) =  t in A & t notin B;
pred ack1 (var1 t, var2 A, B) =  t notin A & t in B;
pred keep1Bit(var1 t,var2 X0)= keepBit(t,X0);
pred keep2Bit(var1 t,var2 X0,X1)=  keep1Bit(t,X0) & keep1Bit(t,X1);
pred keep3Bit(var1 t,var2 X0,X1,X2)=  keep2Bit(t,X0,X1)& keep1Bit(t,X2);
pred keep4Bit(var1 t,var2 X0,X1,X2,X3)= keep2Bit(t,X0,X1) & keep2Bit(t,X2,X3);
pred keep5Bit(var1 t,var2 X0,X1,X2,X3,X4)=
  keep2Bit(t,X0,X1) & keep3Bit(t,X2,X3,X4);
pred keep6Bit(var1 t,var2 X0,X1,X2,X3,X4,X5)=
  keep3Bit(t,X0,X1,X2) & keep3Bit(t,X3,X4,X5);
pred keep7Bit(var1 t,var2 X0,X1,X2,X3,X4,X5,X6)=
  keep3Bit(t,X0,X1,X2) & keep4Bit(t,X3,X4,X5,X6);
pred keep8Bit(var1 t,var2 X0,X1,X2,X3,X4,X5,X6,X7)=
  keep4Bit(t,X0,X1,X2,X3) & keep4Bit(t,X4,X5,X6,X7);
pred keep9Bit(var1 t,var2 X0,X1,X2,X3,X4,X5,X6,X7,X8)=
  keep8Bit(t,X0,X1,X2,X3,X4,X5,X6,X7) & keep1Bit(t,X8);
pred keep12Bit(var1 t,var2 X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11)=
  keep8Bit(t,X0,X1,X2,X3,X4,X5,X6,X7) & keep4Bit(t,X8,X9,X10,X11);
pred unchanged1(var1 t,var2 X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10)=
  keep8Bit(t,X0,X1,X2,X3,X4,X5,X6,X7) & keep3Bit(t,X8,X9,X10);
pred unchanged2(var1 t,var2 X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13)=
  keep12Bit(t,X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11) &
  keep2Bit(t,X12,X13);
pred unchanged3(var1 t,var2 X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,
                     X12,X13,X14,X15,X16,X17)=
  keep12Bit(t,X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11) &
  keep4Bit(t,X12,X13,X14,X15) &
  keep2Bit(t,X16,X17);
pred s_init(var2 S1,S2,Bit) = get(0, S1,S2) & offBit(0,Bit);
pred _sender(var1 t,var2 SBit,S1,S2,SD1,SD2,S2RinD1,S2RinD2,
                  S2RinT1,S2RinT2,R2SoutT1,R2SoutT2) =
 (get(t,S1,S2) &
  send(t+1,S1,S2) &
  keep9Bit(t,SBit,SD1,SD2,S2RinD1,S2RinD2,S2RinT1,S2RinT2,R2SoutT1,R2SoutT2)
 ) |
 (send(t,S1,S2) & mt(t,S2RinT1,S2RinT2) &
  wait(t+1,S1,S2)&
  if(onBit(t,SBit),data1(t+1,S2RinT1,S2RinT2),data0(t+1,S2RinT1,S2RinT2)) &
  transdata(t,SD1,SD2,S2RinD1,S2RinD2) &
  keep3Bit(t,SBit,R2SoutT1,R2SoutT2)
 )|
 (send(t,S1,S2) & ~mt(t,S2RinT1,S2RinT2)  &
  send(t+1,S1,S2) &
   keep9Bit(t,SBit,SD1,SD2,S2RinD1,S2RinD2,S2RinT1,S2RinT2,R2SoutT1,R2SoutT2)
  )|
 (wait(t,S1,S2) &
  ( offBit(t,SBit)  & ack0(t,R2SoutT1,R2SoutT2) |
    onBit(t,SBit)   & ack1(t,R2SoutT1,R2SoutT2)
  )&
  get(t+1,S1,S2) &
  alternateBit(t,SBit) &
  mt(t+1,R2SoutT1,R2SoutT2) &
  getnewData(t+1,SD1,SD2) &
  keep4Bit(t,S2RinD1,S2RinD2,S2RinT1,S2RinT2)) |
 (wait(t,S1,S2)  &
  (onBit(t,SBit)  & ack0(t,R2SoutT1,R2SoutT2) |
   offBit(t,SBit) & ack1(t,R2SoutT1,R2SoutT2) |
   error(t,R2SoutT1,R2SoutT2)
  )&
  send(t+1,S1,S2) &
  mt(t+1,R2SoutT1,R2SoutT2) &
  keep7Bit(t, SBit,SD1,SD2,S2RinD1,S2RinD2,S2RinT1,S2RinT2)
 )|
 (wait(t,S1,S2) & mt(t,R2SoutT1,R2SoutT2)&
 wait(t+1,S1,S2) &
 mt(t+1,R2SoutT1,R2SoutT2) &
 keep7Bit(t,SBit,SD1,SD2,S2RinD1,S2RinD2,S2RinT1,S2RinT2)
 );
pred r_init(var2 R1,R2,RBit)= recieve(0,R1,R2) & offBit(0,RBit);
pred _reciever(var1 t,var2 R1,R2,RBit,RD1,RD2,R2SinT1,
                    R2SinT2,S2RoutT1,S2RoutT2,S2RoutD1,S2RoutD2)=
 ((recieve(t,R1,R2) &
   (offBit(t,RBit)  & data0(t,S2RoutT1,S2RoutT2) |
    onBit (t,RBit)  & data1(t,S2RoutT1,S2RoutT2)
   )) &
  keep3Bit(t,RBit,R2SinT1,R2SinT2) &
  mt(t+1,S2RoutT1,S2RoutT2) &
  deliver(t+1,R1,R2) &
  transdata(t,S2RoutD1,S2RoutD2,RD1,RD2)
  )
|
 (recieve(t,R1,R2) &
   (offBit(t,RBit)  & data1(t,S2RoutT1,S2RoutT2) |
    onBit (t,RBit)  & data0(t,S2RoutT1,S2RoutT2) |
    error(t,S2RoutT1,S2RoutT2)
   )  &
   keep7Bit(t,RBit,R2SinT1,R2SinT2,RD1,RD2,S2RoutD1,S2RoutD2) &
   mt(t+1,S2RoutT1,S2RoutT2) &
   send_ack(t+1,R1,R2))
|
(recieve(t,R1,R2)  & mt(t,S2RoutT1,S2RoutT2) &
 keep5Bit(t,RBit,R2SinT1,R2SinT2,RD1,RD2) &
 mt(t+1,S2RoutT1,S2RoutT2) &
 recieve (t+1,R1,R2))
|
(deliver(t,R1,R2) &
  keep8Bit(t,R2SinT1,R2SinT2,RD1,RD2,S2RoutT1,S2RoutT2,S2RoutD1,S2RoutD2) &
  send_ack(t+1,R1,R2) &
  alternateBit(t,RBit))
|
(send_ack(t,R1,R2) & mt(t,R2SinT1,R2SinT2)   &
 keep7Bit(t,RBit,RD1,RD2,S2RoutT1,S2RoutT2,S2RoutD1,S2RoutD2) &
 recieve (t+1,R1,R2) &
 if(onBit(t,RBit),ack0(t+1,R2SinT1,R2SinT2),ack1(t+1,R2SinT1,R2SinT2)))
|
(send_ack(t,R1,R2) & ~mt(t,R2SinT1,R2SinT2)  &
 keep9Bit(t,RBit,R2SinT1,R2SinT2,RD1,RD2,S2RoutT1,S2RoutT2,S2RoutD1,S2RoutD2) &
 send_ack(t+1,R1,R2));
pred r2s_init(var2 R2SinT1,R2SinT2,R2SoutT1,R2SoutT2) =
  mt(0,R2SinT1,R2SinT2) & mt(0,R2SoutT1,R2SoutT2);
pred _r2s (var1 t,var2 R2SinT1,R2SinT2,R2SoutT1,R2SoutT2)=
 if(
    mt(t,R2SoutT1,R2SoutT2) & ~mt(t,R2SinT1,R2SinT2),
    mt(t+1,R2SinT1,R2SinT2) & (error(t+1,R2SoutT1,R2SoutT2) |
                               transdata(t,R2SinT1,R2SinT2,R2SoutT1,R2SoutT2)),
    keepTag(t,R2SoutT1,R2SoutT2) &  keepTag(t,R2SinT1,R2SinT2)
   );
pred s2r_init(var2 S2RinT1,S2RinT2,S2RoutT1,S2RoutT2) =
  mt(0,S2RinT1,S2RinT2) & mt(0,S2RoutT1,S2RoutT2);

pred _s2r(var1 t,var2 S2RinT1,S2RinT2,S2RoutT1,S2RoutT2,S2RoutD1,S2RinD1,
                      S2RinD2,S2RoutD2) =
    if(
       mt(t,S2RoutT1,S2RoutT2) & ~mt(t,S2RinT1,S2RinT2),
       (error(t+1,S2RoutT1,S2RoutT2)|
        transdata(t,S2RinT1,S2RinT2,S2RoutT1,S2RoutT2)) &
       mt(t+1,S2RinT1,S2RinT2) &
       transdata(t,S2RinD1,S2RinD2,S2RoutD1,S2RoutD2),

      keep6Bit(t,S2RoutT1,S2RoutT2,S2RinT1,S2RinT2,S2RoutD1,S2RoutD2)
      )
    &     keep2Bit(t,S2RinD1,S2RinD2);

var2
   P1 , P2,
   SBit ,
   S1, S2,
   SD1, SD2,
   RBit ,
   R1, R2,
   RD1, RD2,
   S2RinD1, S2RinD2,
   S2RoutD1, S2RoutD2,
   S2RinT1, S2RinT2,
   S2RoutT1, S2RoutT2,
   R2SinT1, R2SinT2,
   R2SoutT1, R2SoutT2;

pred init()   =
 0 notin P1 & 0 notin P2                 &
 s_init(S1,S2,SBit)                        &
 r_init(R1,R2,RBit)                        &
 r2s_init(R2SinT1,R2SinT2,R2SoutT1,R2SoutT2) &
 s2r_init(S2RinT1,S2RinT2,S2RoutT1,S2RoutT2) ;

pred sender(var1 t) =
 (t notin P1 & t notin P2) &
 _sender(t,SBit,S1,S2,SD1,SD2,S2RinD1,S2RinD2,S2RinT1,S2RinT2,R2SoutT1,R2SoutT2) &
 unchanged1(t,RBit,R1,R2,RD1,RD2,S2RoutD1,S2RoutD2,S2RoutT1,S2RoutT2,R2SinT1,
            R2SinT2);

pred sender_bug(var1 t) =
 _sender(t,SBit,S1,S2,SD1,SD2,S2RinD1,S2RinD2,S2RinT1,S2RinT2,R2SoutT1,R2SoutT2) &
 unchanged1(t,RBit,R1,R2,RD1,RD2,S2RoutD1,S2RoutD2,S2RoutT1,S2RoutT2,R2SinT1,
            R2SinT2);


pred reciever(var1 t) =
 (t notin P1 & t in P2) &
 _reciever(t,R1,R2,RBit,RD1,RD2,R2SinT1,R2SinT2,S2RoutT1,S2RoutT2,S2RoutD1,
           S2RoutD2) &
 unchanged1(t,SBit,S1,S2,SD1,SD2,S2RinT1,S2RinT2,S2RinD1,S2RinD2,R2SoutT1,
            R2SoutT2);

pred s2r(var1 t)=
 (t in P1 & t notin P2) &
 _s2r(t,S2RinT1,S2RinT2,S2RoutT1,S2RoutT2,S2RinD1,S2RinD2,S2RoutD1,S2RoutD2)&
 unchanged2(t,SBit,S1,S2,SD1,SD2,RBit,R1,R2,RD1,RD2,R2SinT1,R2SinT2,R2SoutT1,
           R2SoutT2);

pred r2s(var1 t)=
 (t in P1 & t in P2) &
 _r2s (t,R2SinT1,R2SinT2,R2SoutT1,R2SoutT2)&
 unchanged3(t,SBit,S1,S2,SD1,SD2,RBit,R1,R2,RD1,RD2,S2RinD1,S2RinD2,S2RoutD1,
           S2RoutD2,S2RinT1,S2RinT2,S2RoutT1,S2RoutT2);

pred trans () = all1 t: t<$ =>sender(t)|reciever(t)|s2r(t)|r2s(t);

pred abp () = init() & trans();

#### Properties
pred phi1() =
 all1 s,t: (s<t & t<$ & get(s,S1,S2) & get(t,S1,S2) &
           all1 q: s< q & q<t => ~get(q,S1,S2))
          => all1 q: s< q & q<t => (onBit(s+1,SBit) <=> onBit(q,SBit));

pred phi2() =
all1 s,t: (s<t  & t<$ & offBit(s,SBit) & onBit(t,SBit)
          &  all1 q: s<= q & q<t => offBit(q,SBit))
          =>  ex1 r: s<= r & r<t & ack0(r,R2SoutT1,R2SoutT2);


pred phi3() =
all1 s,t: (s<t  & t<$ & onBit(s,SBit) & offBit(t,SBit)
          &  all1 q: s<= q & q<t => onBit(q,SBit))
          =>  ex1 r: s<= r & r<t & ack1(r,R2SoutT1,R2SoutT2);
```

C.4 Bus Arbiter Protocol for Three Cells

Below is the M2L-Str formalization of the bus arbiter protocol for three cells.

```
m2l-str;

pred atmost_one  (var0 a,b,c)  =  (a => ~ (b |c)) & (b => ~(a|c)) & (c => ~ (a|b));
pred exactly_one (var0 a,b,c)  =  (a|b|c) & atmost_one(a,b,c);

pred Celle(var0 token, n_token, wait, n_wait, request, token_in,
                token_out, garant_in, garant_out, overrid_in, overrid_out, ack)=
 (n_token     <=> token_in) &
 (n_wait      <=> ((wait | token) & request)) &
 (ack         <=> ((wait & token | garant_in)& request)) &
 (token_out   <=> token) &
 (garant_out  <=> ((~ request)& garant_in)) &
 (overrid_out <=> (wait & token | overrid_in));

pred init_celle0(var2 T,W) = 0 in T    & 0 notin W ;
pred init_celle(var2 T,W)  = 0 notin T & 0 notin W ;
pred first_last_wiring(var2 Tin0,Tin1,Tin2,Tout0,Tout1,Tout2,
                            Oin0,Oin1,Oin2,Oout0,Oout1,Oout2,
                            Gin0, Gin1,Gin2,Gout0,Gout1,Gout2) =
      all1 t:
         (t in Oout0 <=> t notin Gin0) &
         (t in Oin0 <=> t  in Oout1)   &
         (t in Oin1 <=> t  in Oout2)   &
         (t notin Oin2)                &
         (t in Gout0 <=> t in Gin1)    &
         (t in Gout1 <=> t in Gin2)    &
         (t in Tout0 <=> t in Tin1)   &
         (t in Tout1 <=> t in Tin2)   &
         (t in Tout2 <=> t in Tin0);


pred arbiter(var2 R0,R1, R2,A0, A1, A2,
             T0,T1,T2,W0,W1,W2,Tin0,Tin1,Tin2,Tout0,Tout1,Tout2,Oin0,Oin1,
             Oin2,Oout0,Oout1,Oout2,Gin0, Gin1,Gin2,Gout0,Gout1,Gout2) =

 init_celle0(T0,W0) &
 init_celle(T1,W1) &
 init_celle(T2,W2) &
 first_last_wiring(Tin0,Tin1,Tin2,Tout0,Tout1,Tout2,Oin0,Oin1,Oin2,
               Oout0,Oout1,Oout2,Gin0, Gin1,Gin2,Gout0,Gout1,Gout2) &
 all1 t: t <$ =>
  Celle(t in T0,t+1 in T0,t in W0,t+1 in W0,t in R0,t in Tin0, t in Tout0,
        t in Gin0, t in Gout0, t in Oin0, t in Oout0,t in A0)
& Celle(t in T1,t+1 in T1,t in W1,t+1 in W1,t in R1,t in Tin1, t in Tout1,
        t in Gin1, t in Gout1, t in Oin1, t in Oout1,t in A1)
& Celle(t in T2,t+1 in T2,t in W2,t+1 in W2,t in R2,t in Tin2, t in Tout2,
        t in Gin2, t in Gout2, t in Oin2, t in Oout2,t in A2);

pred spec(var2 R0,R1,R2,A0,A1,A2,T0,T1,T2) =
             (all1 t: t<$  => exactly_one(t in T0, t in T1,t in T2))
           & (all1 t: t<$  => atmost_one (t in A0,t in A1,t in A2))
           & (all1 t: t<$ & t in A0 => ex1 p : p<=t & p in R0 &
                       all1 x: p<x & x<t => x notin R0 & x notin A0)
           & (all1 t: t<$ & t in A1 => ex1 p : p<=t & p in R1 &
                       all1 x: p<x & x<t =>  x notin R1 & x notin A1)
           & (all1 t: t<$ & t in A2 => ex1 p : p<=t & p in R2 &
                       all1 x: p<x & x<t =>  x notin R2 & x notin A2);
```

Bibliography

[AB00] Abdelwaheb Ayari and David Basin. Bounded model construction for monadic second-order logics. In *12th International Conference on Computer-Aided Verification (CAV'00)*, number 1855 in LNCS. Springer-Verlag, 2000.

[ACW90] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, April 1990.

[AEFM89] K. R. Abrahamson, J. A. Ellis, M. R. Fellows, and M. E. Mata. On the complexity of fixed parameter problems (extended abstract). In *30th Annual Symposium on Foundations of Computer Science.* IEEE, 1989.

[AKPS94] H. Ait-Kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1-2):263–283, January 1994.

[Ard61] Dean N. Arden. Delayed-logic and finite-state machines. In *Proceedings of the Second Annual Symposium and Papers from the First Annual Symposium on Switching Circuit Theory and Logical Design*, pages 133–151. American Institute of Electrical Engineers, 1961.

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

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

[BAMP81] M. Ben-Ari, Z. Manna, and Amir Pnueli. The temporal logic of branching time. In *Eighth Annual ACM Symposium on Principles of Programming Languages*, volume 20, pages 207–226. ACM, ACM, 1981.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *LNCS.* Springer, 1999.

[BCDM86] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers*, C-35(12):1034–1044, December 1986.

[BF98] David Basin and Stefan Friedrich. Combining WS1S and HOL. In *Frontiers of Combining Systems, Second International Workshop, Amsterdam, September 1998*, Applied Logic Series. Kluwer Academic Publishers, 1998. To appear.

[BF00a] J. Bodeveix and M. Filali. Fmona: A tool for expressing validation techniques over infinite state systems. In S.Graf and M. Schwartzbach, editors, *TACAS'00*, volume 1785 of *Lecture Notes in Computer Science*, pages 204–219. Springer, 2000.

[BF00b] Jean-Paul Bodeveix and Mamoun Filali. Experimenting acceleration methods for the validation of infinite state systems. In *ICDCS Workshop on Distributed System Validation and Verification*, pages E23–E30, 2000.

[BFGP02] David Basin, Stefan Friedrich, Marek Gawkowski, and Joachim Posegga. Bytecode Model Checking: An Experimental Analysis. In Dragan Bosnacki and Stefan Leue, editors, *Model Checking Software, 9th International SPIN Workshop*, volume 2318 of *LNCS*, pages 42–59, Grenoble, France, April 2002. Springer-Verlag.

[BK95] D. A. Basin and N. Klarlund. Hardware verification using monadic second-order logic. *Lecture Notes in Computer Science*, 939:31–41, 1995.

[BK98] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *The Journal of Formal Methods in Systems Design*, 13(3), November 1998.

[BKR97] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA '96, London, Ontario, Canada, LNCS 1260.* Springer Verlag, 1997.

[BL80] J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, January 1980.

[BL94] Hans Kleine Büning and Theodor Lettmann. *Aussagenlogik: Deduktion und Algorithmen.* B. G. Teubner, Stuttgart, 1994.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. *Lecture Notes in Computer Science*, 1427:319–331, 1998.

[Boy92] Thierry Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4), October 1992.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

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

[BS93] Rolf Backofen and Gert Smolka. A complete and recursive feature theory. In *Proceedings of the 31st ACL*, pages 193–200, Columbus, Ohio, 1993. ACL. A full version has appeared as Research Report RR-92-30, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6, 1960.

[Büc62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Int. Congr. for Logic, Methodology, and Philosophy of Science*. Stanford Univ. Press, 1962.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986. An early version appeared in *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983.

[CGH$^+$93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 15–30, Amsterdam, The Netherlands, April 1993. North-Holland.

[CGL94] E. Clarke, D. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGM86] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs.* North Holland, 1986.

[CGS98] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified Boolean formulae. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, July 26–30 1998.

[Chu62] Alonzo Church. Logic, arithmetic and automata. In Almqvist and Wiskells, editors, *Proceedings of the International Congress of Mathematicians*, pages 23–35, Uppsala 1963, 1962.

[CJEF96] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):77–104, August 1996.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[Coh96] Richard M. Cohen. The Defensive Java Virtual Machine Specification Version, Alpha 1 Release. Technical report, Computational Logic, Inc; http://www.cli.com/software/djvm/html-0.5/d jvm-report.html, 1996.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 3–5 1971 1971.

[Dij72] E. W. Dijkstra. Notes on Structured Programming. In O.-J. Dahl, E. W. Disjkstra, and C. A. R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[Dil88] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1988.

[DKS99] Niels Damgaard, Nils Klarlund, and Michael I. Schwartzbach. YakYak: Parsing with logical side constraints. In *Proceedings of DLT'99*, 1999.

[Don65] J. E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965.

[Don70] J. E. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451, 1970.

[EE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[Elg61] C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the AMS*, 98, 1961.

[EMS00] Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proceedings of European Symposium on Programming Languages and Systems*, volume 1782 of *LNCS*. Springer Verlag, 2000.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th International Computer Aided Verification Conference*, pages 463–478, 1993.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[GNT01a] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for quantified boolean logic satisfiability. In *Proceedings of the 17th International Conference on Artificial Intelligence (IJCAI-01)*, August 4–10 2001.

[GNT01b] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE: A system for deciding Quantified Boolean Formulas Satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01)*, June 2001.

[Gor86] M. J. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.

[Gor93] M. J. C. Gordon. *Introduction to HOL: A Theorem Proving Environment*. Cambridge University Press, 1993.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. 5th International Computer Aided Verification Conference*, pages 438–449, 1993.

[GPVW95] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[GW00] Jan Friso Groote and Joost P. Warners. The propositional formula checker HeerHugo. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT20000: Highlights of Satisfiability Research in the year 2000*, Frontiers in Artificial Intelligence and Applications. Kluwer Academic, 2000.

[HH01] B. Hirsch and U. Hustadt. Translating PLTL into WS1S: Application description. *In Methods for Modalities II. University of Amsterdam*, 2001.

[HJJ$^+$96] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95*, volume 1019 of *LNCS*. Springer Verlag, 1996.

[HJJK95] J. G. Henriksen, J. Jensen, M. Joergensen, and N. Klarlund. MONA: Monadic second-order logic in practice. *Lecture Notes in Computer Science*, 1019:89–101, 1995.

[HK90] Zri Har'El and Robert Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, January 1990.

[HK91] Z. Har'el and R. P. Kurshan. Automatic verification of coordinating systems. In *Proc. of Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1991.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hol97] Gerard J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–95, May 1997.

[ID93] C. N. Ip and D. L. Dill. Better verification through symmetry. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 97–112, Amsterdam, The Netherlands, April 1993. North-Holland.

[JTI94] D. Johnson, M. Trick, and D. Implementation. Dimacs series in discrete mathematics and theoretical computer science, 1994.

[Kam68a] H. W. Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, University of California, Los Angeles, 1968.

[Kam68b] Hans Kamp. *On tense logic and the theory of order.* PhD thesis, UCLA, 1968.

[KKS96] N. Klarlund, J. Koistinen, and M.I. Schwartzbach. Formal design constraints. In *Proceedings of OOPSLA '96*, 1996.

[Kla99] N. Klarlund. A theory of restrictions for logics and automata. In *CAV '99*, volume 1633 of *LNCS*. Springer, 1999.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[KMMG97] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. MOSEL: A flexible toolset for monadic second-order logic. *Lecture Notes in Computer Science*, 1217:183–195, 1997.

[KMMP97] Y. Kesten, O. Maler, M. Marcus, and A. Pnueli. Symbolic model checking with rich assertional languages. *Lecture Notes in Computer Science*, 1254:424–435, 1997.

[KNS96] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in verification based on trace abstractions. *Lecture Notes in Computer Science*, 1169:341–353, 1996.

[Kur89] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, New York, May 1989. Springer-Verlag.

[Kur94] R. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[Lei81a] E. Leiss. On generalized language equations. *Theoretical Computer Science*, 14(1):63–77, April 1981.

[Lei81b] E. Leiss. Succinct representation of regular languages by Boolean automata. *Theoretical Computer Science*, 13(3):323–330, March 1981.

[Ler01] Xavier Leroy. Java bytecode verification: An overview. In *Computer Aided Verification, 13th International Conference*, volume 2001 of *LNCS*, pages 265–285, Paris, France, July 2001. Springer-Verlag.

[Let01] Reinhold Letz. Advances in decision procedures for quantified boolean formulas. In Uwe Egly, Rainer Feldmann, and Hans Tompits, editors, *Proceedings of QBF2001 workshop at IJCAR'01*, June 2001.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM, ACM, January 1985.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In Rohit Parikh, editor, *Proceedings of the Conference on Logic of Programs*, volume 193 of *LNCS*, pages 196–218, Brooklyn, NY, June 1985. Springer.

[Mar85] Alain J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5(3):265–276, October 1985.

[MBBC95] Z. Manna, N. Bjoerner, A. Browne, and E. Chang. STeP: The Stanford Temporal Prover. In *TAPSOFT'95: Theory and Practice of Software Development*, volume 915, 1995.

[MC97] F. Morawietz and T. Cornell. On the recognizability of relations over a tree definable in a monadic second order tree description language. Research Report SFB 340-Report 85, Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft, 1997.

[McM92] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992. CMU-CS-92-131.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, October 1966.

[Mey75] A. Meyer. Weak monadic second-order theory of successor is not elementary-recursive. In *LOGCOLLOQ: Logic Colloquium*, volume 453. Springer-Verlag, 1975.

[MMZ$^+$01] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 1–12, Department of Computer Science, University of Aarhus, May 1995. BRICS. vi+334pp.

[Möd01] Sebastian Mödersheim. Modellierung und Analyse sequentieller Hardware in monadischen Logiken zweiter Stufe. Master's thesis, Albert-Ludwigs-Universität Freiburg, 2001.

[MS73] A. R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *ACM Symposium on Theory of Computing*, New York, April 1973. ACM Press.

[MS91] K. L. McMillan and J. Schwalbe. Formal verification of the encore gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, pages 242–251. (sponsored by Information Processing Society, Tokyo, Japan), 1991.

[MW97] William McCune and Larry Wos. Otter—the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, April 1997.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is Type-Safe — Definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages.* ACM Press, 1998.

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6. Elsevier Science B.V., 2001.

[ORRS96] S. Owre, S. Rajah, J. M. Rushby, and N. Shankar. PVS: combining specification, proof checking, and model checking. *Lecture Notes in Computer Science*, 1102:411–??, 1996.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1994.

[PBZ02] David A. Plaisted, Armin Biere, and Yunshan Zhu. A satisfiability procedure for quantified boolean formulae. *Discrete Applied Mathematics*, August 2002.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Lecture Notes in Computer Science*, 818:377–390, 1994.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.

[PS00] Amir Pnueli and Elad Shahar. Liveness and acceleration in parameterized verification. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science.* Springer, 2000.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, Lecture Notes in Computer Science, Vol. 137, pages 337–371. SV, Berlin/New York, 1982.

[Rab69] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer.Math.Soc.*, 141:1–35, 1969.

[Rin99a] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10, 1999.

[Rin99b] Jussi Rintanen. Improvements to the evaluation of quantified boolean formulae. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)*. Morgan Kaufmann Publishers, S.F., July 31–August 6 1999.

[Rin01] Jussi Rintanen. Partial implicit unfolding in the Davis-Putnam procedure for quantified Boolean formulae. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *LNCS*. Springer-Verlag, Berlin, 2001.

[Rog94] James Rogers. *Studies in the logic of trees with applications to grammar formalisms.* PhD thesis, University of Delaware, Pittsburgh, PA, 1994.

[Sal69] A. Salomaa. *Theory of Automata*, pages 120–123. Pergamon Press, 1969.

[SB01] Christoph Scholl and Bernd Becker. Checking equivalence for partial implementations. In *Design Automation Conference*, 2001.

[SC85] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32:733–749, 1985.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium, San Diego, California, 19–21 January 1998*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[Sht00] Ofer Shtrichman. Tuning sat checkers for bounded model checking. In *12th International Conference on Computer-Aided Verification (CAV'00)*, number 1855 in LNCS. Springer-Verlag, 2000.

[Slu85] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41(2-3):305–318, 1985.

[Sta89] Gunnar Staalmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Technical report, European Patent Nr. 0403 454 (1995), US Patent Nr. 5 276 897, Swedish Patent Nr. 467 076 (1989), 1989.

[Str94] H. Straubing. *Finite automata, formal logic, and circuit complexity.* Birkhäuser, 1994.

[TB73] B. A. Trachtenbrot and Ya. M. Barzdin. *Finite Automata - Behavior and Synthesis.* North-Holland, Amsterdam, 1973.

[Tha73] James W. Thatcher. Tree automata: An informal survey. In *Alfred V. Aho, Currents in the Theory of Computing.* Prentice-Hall, 1973.

[Tho89] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), North-Holland, 1989.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4. Elsevier Science Publishers B. V., 1990.

[Tho97] W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer-Verlag, Berlin, 1997.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem in second-order logic. *Math. Systems Theory*, 2:57–81, 1968.

[UAH74] J. D. Ullman, Alfred V. Aho, and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science*, 1043:238–266, 1996.

[Vor96] S. Vorobyov. An improved lower bound for the elementary theories of trees. *Lecture Notes in Computer Science*, 1104:275–287, 1996.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *lics86*, pages 332–344, 1986.

[WAB$^+$99] Christoph Weidenbach, Bijan Afshordel, Uwe Brahm, Christian Cohrs, Thorsten En gel, Enno Keen, Christian Theobalt, and Dalibor Topić. System description: SPASS version 1.0.0. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16-99)*, volume 1632 of *LNAI*, pages 378–382, Berlin, July 7–10 1999. Springer.

[WBCG00] Poul Williams, Armin Biere, Edmund Clarke, and Anubhav Gupta. Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking. In *Proceedings of CAV'00*, number 1855 in LNCS. Springer-Verlag, 2000.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM, ACM, January 1986.

[WP89] A. Wilk and A. Pnueli. Specification and verification of VLSI systems. In *IEEE/ACM International Conference on Computer-Aided Design*, 1989.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *CADE'97*, volume 1249 of *LNAI*. Springer, 1997.

[ZM88] Ramin Zabih and David McAllester. A rearrangement search strategy for determining propositional satisfiability. In Tom M. Smith, Reid G.; Mitchell, editor, *Proceedings of the 7th National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. Morgan Kaufmann.

[Zuc86] L. Zuck. *Past Temporal Logic.* PhD thesis, Weizmann Institute, Rehovot, Israel, 1986.