RICE UNIVERSITY

Explicit or Symbolic Translation of Linear Temporal Logic to Automata
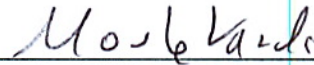
by

Kristin Yvonne Rozier
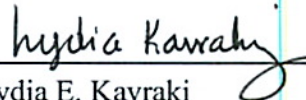
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree
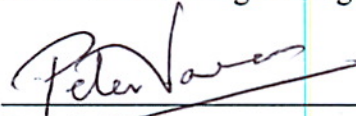
Doctor of Philosophy

Approved, Thesis Committee:

_____

Moshe Y. Vardi, Chair
Karen Ostrum George Professor in
Computational Engineering

_____

Lydia E. Kavraki
Noah Harding Professor of Computer
Science and Bioengineering

_____

Peter J. Varman
Professor of Electrical and Computer
Engineering

Houston, Texas

April, 2012

ABSTRACT

Explicit or Symbolic Translation of Linear Temporal Logic to Automata

by

Kristin Yvonne Rozier

Formal verification techniques are growing increasingly vital for the development of safety-critical software and hardware in practice. Techniques such as requirements-based design and model checking for system verification have been successfully used to verify systems for air traffic control, airplane separation assurance, autopilots, CPU logic designs, life-support, medical equipment, and other functions that ensure human safety.

Formal behavioral specifications written early in the system-design process and communicated across all design phases increase the efficiency, consistency, and quality of the system under development. We argue that to prevent introducing design or verification errors, it is crucial to test specifications for *satisfiability*. We advocate for the adaptation of a new sanity check via satisfiability checking for property assurance. Our focus here is on specifications expressed in Linear Temporal Logic (LTL). We demonstrate that LTL satisfiability checking reduces to model checking and satisfiability checking for the specification, its complement, and a conjunction of all properties should be performed as a first step to LTL model checking.

We report on an experimental investigation of LTL satisfiability checking. We introduce a large set of rigorous benchmarks to enable objective evaluation of LTL-to-automaton algorithms in terms of scalability, performance, correctness, and size of the automata produced. For explicit model checking, we use the Spin model checker; we tested all LTL-

to-explicit automaton translation tools that were publicly available when we conducted our study. For symbolic model checking, we use CadenceSMV, NuSMV, and SAL-SMC for both LTL-to-symbolic automaton translation and to perform the satisfiability check. Our experiments result in two major findings. First, scalability, correctness, and other debilitating performance issues afflict most LTL translation tools. Second, for LTL satisfiability checking, the symbolic approach is clearly superior to the explicit approach.

Ironically, the explicit approach to LTL-to-automata had been heavily studied while only one algorithm existed for LTL-to-symbolic automata. Since 1994, there had been essentially no new progress in encoding symbolic automata for BDD-based analysis. Therefore, we introduce a set of 30 symbolic automata encodings. The set consists of novel combinations of existing constructs, such as different LTL formula normal forms, with a novel transition-labeled symbolic automaton form, a new way to encode transitions, and new BDD variable orders based on algorithms for tree decomposition of graphs. An extensive set of experiments demonstrates that these encodings translate to significant, sometimes exponential, improvement over the current standard encoding for symbolic LTL satisfiability checking.

Building upon these ideas, we return to the explicit automata domain and focus on the most common type of specifications used in industrial practice: safety properties. We show that we can exploit the inherent determinism of safety properties to create a set of 26 explicit automata encodings comprised of novel aspects including: state numbers versus state labels versus a state look-up table, finite versus infinite acceptance conditions, forward-looking versus backward-looking transition encodings, assignment-based versus BDD-based alphabet representation, state and transition minimization, edge abbreviation, trap-state elimination, and determinization either on-the-fly or up-front using the subset construction. We conduct an extensive experimental evaluation and identify an encoding that offers the best performance in explicit LTL model checking time and is constantly faster than the previous best explicit automaton encoding algorithm.

Acknowledgments

I have been lucky to have not one but three academic fathers:

> To my father, Michael Alexander Roberts, for passing down the computer scientist gene and having such an infectious passion for this discipline that I couldn't help falling in love. I could not have done this without your early support and encouragement in everything from helping me to build computers as soon as I had the manual dexterity to helping me find a path I was truly passionate about. I love you.

> To my M.S. adviser, Paul K. Stockmeyer, for introducing me to the wonders of automata theory and encouraging me to pursue it, especially when I doubted myself. Thank you so much for helping me realize my dream and advising me to pursue my PhD in formal methods – I am so glad I listened to you, about so many things.

> To my Ph.D. adviser, Moshe Y. Vardi, for guiding me, inspiring me, channelling my love of automata theory into a PhD thesis, helping me to become a better researcher, and teaching me so many lessons along the way. It is an honor to be your student. I hope I have made you proud.

To the Computer Science Department at the College of William and Mary, a true academic family, for giving me a solid foundation in theoretical computer science and supporting me every step of the way as I built upon it. I am indebted to all of you, especially Bill Bynum, Gianfranco Ciardo, Stefan Feyock, Phil Kearns, Lawrence Leemis, Debbie Noonan, Robert Noonan, Stephen Park, Richard Prosl, Zia-ur Rahman, Evgenia Smirni, Andreas Stathopoulos, Paul Stockmeyer, and Virginia Torczon.

Contents

Illustrations

Tables

Chapter 1

Introduction

Safety-critical systems have become ubiquitous in our everyday lives and grow increasingly complex every year. They fly our planes, control our nuclear power plants, run our medical devices, and so much more. Yet, how do we know they are safe? We can use *formal methods* for the design, specification, and verification of life-critical hardware and software systems. *Formal methods* refers to a collection of verification techniques, all of which entail the utilization of mathematical logic, to provide checks of correctness with a very high level of assurance. *Verification* of a software or hardware system involves checking whether the system in question behaves as it was designed to behave; i.e. it answers the question, "Does the system as we have designed it have the intended emergent behaviors?" (If it does not, it is desirable to find out early in the design process!) *Design Validation* involves checking whether a system design incorporates the system requirements; i.e. it answers the question, "Does the logical system representation accurately encapsulate the system design?" These tasks, system verification and design validation, can be accomplished thoroughly and reliably using formal methods.

Due to its automated nature and ease of use, *model checking* [6, 7] has become one of the most widely-used formal methods. Model checking is the formal process through which a desired behavioral property (the specification) is verified to hold for a given system (the model) via an exhaustive enumeration (either explicit or symbolic) of all reachable system states and the behaviors that cause the system to transition between them. If the specification is found not to hold in all system executions, a *counterexample* is produced, consisting

of a trace of the model from a start state to an error state in which the specification is violated, providing a very helpful tool for debugging the system design; see Figure 1.1. Model checking has enjoyed broad industrial adaptation and has been successfully used to verify systems such as air traffic control, airplane separation assurance, autopilots, CPU designs, life-support systems, medical equipment (such as devices that administer radiation), and many other systems that ensure human safety. Since model checking requires the writing of formal properties, it has also helped fuel the industrial adoption of *property-based design*, wherein formal specifications are written early in the system-design process and communicated across all design phases [8, 9, 10, 11, 12].
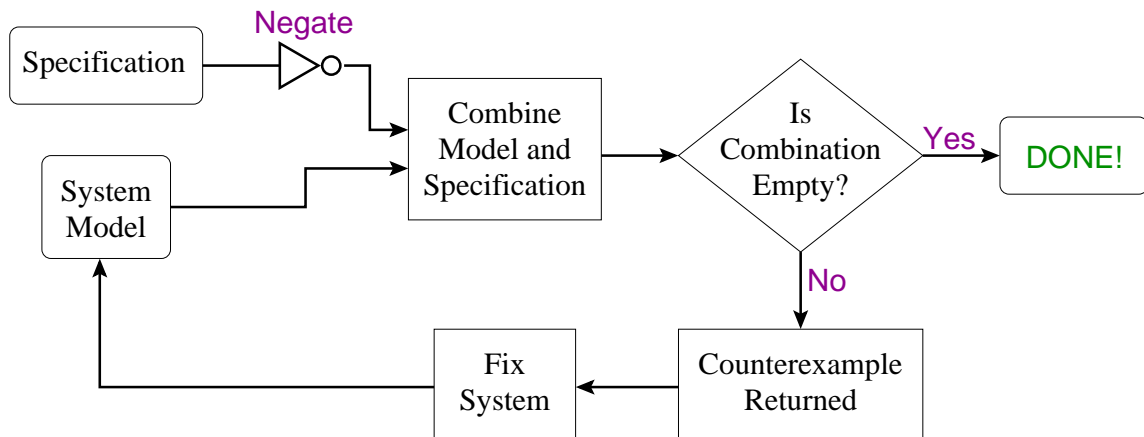
Figure 1.1 : System diagram illustrating the model checking process.

For example, in the field of aeronautics, several full-scale air traffic control systems have been successfully verified using model checking techniques stemming from those we discuss here. Symbolic model checking [13] in SMV [14] was used to verify the system requirements specification for the Traffic Alert and Collision Avoidance System (TCAS II), an air traffic guidance system required onboard large commercial aircraft [15]. Properties checked included the absence of undesirable nondeterminism, mutual exclusion, termina-

tion, absence of references to undefined parameters, and elimination of inconsistencies in the protocol specification. SMV also enabled verification of the A-7E aircraft software requirements to ensure internal aircraft modes were consistently enabled so that procedures for monitoring the aircraft's windspeed, velocity, and alignment accurately contribute to the aircraft's calculated relative position [16]. The Small Aircraft Transportation System (SATS), which represents an approach to air traffic management for non-towered non-radar airports in the United States, was model checked to verify the absence of deadlocks, that aircraft separation is maintained, and that the system is robust in the face of unexpected rare events such as equipment failures or aircraft deviating off-course [17]. A prototype for NASA's Tactical Separation Assured Flight Environment (TSAFE), which provides aircraft conflict detection and resolution, was also model checked to verify the absence of synchronization faults, employing compositional verification techniques [18].

Rockwell Collins and the University of Minnesota developed a translator framework, in coordination with NASA's Aviation Safety Program, that was successfully used to verify many complex industrial avionics systems [19]. The ADGS-2100 Adaptive Display and Guidance System Window Manager is a Rockwell Collins product that ensures the data from different applications is routed to the correct heads-down and heads-up displays and performs display management in next-generation commercial aircraft. NuSMV [20], via this translator framework, was used early in the design process to ensure that the ADGS-2100 does not contain logic errors that could make critical flight information unavailable to the flight crew [21]. Similarly, NuSMV was utilized in verifying that the mode logic of the FCS 5000, a new family of Flight Control Systems, met system requirements [22]. The translator framework was also critical in the verification for Phase I and Phase II of the Certification Technologies for Advanced Flight Critical Systems (Cer-TA FCS) program of the U.S. Air Force Research Laboratory (AFRL) [23]. For Phase I, NuSMV was used

to check against Operational Flight Program (OFP) requirements for an unmanned aerial vehicle, resulting in significant modifications to the final system to address errors uncovered in the original Redundancy Management logic that were not uncovered by testing. For Phase II, Prover [24] was used to verify whether the six actuator commands for the aircraft's control surfaces would always be within dynamically computed upper and lower limits, indicating design errors for another UAV.

1.1   Motivation

The time-honored techniques of simulation and testing also address similar questions and are extremely useful debugging tools in early stages of system design and verification. Both of these methods involve checking the system's behavior on a large, but rarely exhaustive, set of expected inputs. However, as a system is refined, the remaining bugs become fewer and more subtle and require more time to uncover. A major gap in the process of using simulation and/or testing for verification and validation is that there is no way to tell when these techniques are finished, i.e., when all of the bugs in the system have been found. In other words, testing and simulation can be used to demonstrate the presence of bugs but not the absence of bugs. It is not even possible to accurately estimate how many bugs remain [25]. Another open question is that of *coverage*, of both the possible system inputs and the system state space. Coverage refers, respectively, to the completeness of the set of system inputs or to the completeness of the specification, such that a set of inputs or a system state is considered "uncovered" if it is not essential to the success of the verification procedure [26, 27]. Quite simply, it has been proved that testing and simulation cannot be used to guarantee an ultra-high level of reliability within any realistic period of time [28]. For some systems, this is an acceptable risk. For these systems, it is enough to reduce the bug level below a certain measurable tolerance, for example in terms of frequency

in time. For *safety-critical* systems, or other systems, such as financial systems, where reliability is key because failures can be potentially catastrophic, we require an absolute assurance that the system follows its specification via an examination of all possible behaviors, including those that are unexpected or unintended. This assurance is provided by a *design-for-verification* paradigm incorporating property-based design and model checking. Design-for-verification refers to a method for incorporating verification as a top consideration of the design process via extending software design best practices to enable better automated verification [29, 30, 31, 32].

While there are a range of different techniques for formal verification, model checking is particularly well-suited to the automated verification of finite-state software and hardware systems [33]. Once the system model and specification have been determined, the performance of the model-checking step is often very fast, frequently completing within minutes [25]. The counterexample returned in the case where a bug is found provides necessary diagnostic feedback. Furthermore, iterative refinement and re-checking of the failed specification can provide a wealth of insight into the detected faulty system behavior. Model checking lends itself to integration into industrial design life-cycles as the learning curve is quite shallow and easily outweighed by the advantages of early fault detection. The required levels of user interaction and specialized expertise needed to effectively utilize a model checker are minimal compared to other methods of formal verification. Moreover, partial specifications can be checked, allowing verification steps to occur intermittently throughout system design. However, there is a trade-off between the high level of automation provided by model checking and the high level of expressiveness and control that may be required for verification in some cases. For this reason, certain systems benefit from the use of alternative verification techniques, such as theorem proving, which involves proving using formal deduction that the formal system model implies the desired properties. Still,

model checking's high level of automation makes it a preferable verification method where applicable since the performance time and quality of insight obtained from a negative result when using theorem proving for verification are highly dependent on the particular skill set of the person providing the proof.

Currently, using a design-for-verification paradigm incorporating property-based design and model checking for verification requires significantly more time and higher cost up front than simulations or other forms of testing (with some exceptions, e.g. [23]) but the significantly higher level of assurance provided by formal verification outweighs the initial cost of implementation in the case of systems where human life or safety is at risk. Recent algorithmic advances have vastly increased the size and complexity of the systems that we can analyze using formal verification methods. (Examples include partial order reduction [34], the use of bisimulation equivalences [35], compositional verification [36], bounded model checking [37], and "lite" formal methods such as static analysis [38].) However, today's verification techniques do not scale with the ever-increasing complexity and diversity of modern life critical systems. There is an urgent need to extend research in model checking and property-based design for the verification of ever more challenging problem domains.

1.2   Model Checking

Formally, the technique of model checking checks that a system, starting at a start state, satisfies a specification. Let M be a state-transition graph representing the system with a set of states S and $s \in S$ as the start state. Let φ be the specification in temporal logic. We check that $M, s \models \varphi$. In other words, we check that M satisfies ("models") φ. This technique of temporal logic model checking was developed independently by Clarke and Emerson [6] in 1981 and Quielle and Sifakis [7] in 1982. Thus, 1981 is considered the

Description:	Implementation:
1. Create a mathematical model of the system.	1. Define the system model M containing traces over the set *Prop* of propositions.
2. Encapsulate desired properties in a formal specification.	2. Let specification φ be a formula over the set *Prop*.

3. Check that the model satisfies the specification.	3. Check that $M \models \varphi$:
	
	• Translate the specification $\neg\varphi$ into a Büchi automaton $A_{\neg\varphi}$ and compose it with the system model M to form $A_{M,\neg\varphi}$.
	
	• Check $A_{M,\neg\varphi}$ for *nonemptiness*. That is, search for a trace that is accepted by $A_{M,\neg\varphi}$.
	
	  – If such a trace exists, return it as a *counterexample*.
	  – If no such trace exists, return *TRUE*.

Table 1.1 : Definition of Model Checking.

birth year of model checking.

In the *automata-theoretic approach* to model checking [39], we first complement the specification, φ. Then $\neg\varphi$ is translated into an automaton, $A_{\neg\varphi}$, that accepts exactly all computations that satisfy the formula $\neg\varphi$. $A_{\neg\varphi}$ is then composed with the model M under verification, forming $A_{M,\ \neg\varphi}$ [40]. Intuitively, any accepting path in $A_{M,\ \neg\varphi}$ represents a case where the system M allows a behavior that violates the specification φ. The model checker then searches for such a trace of the model that is accepted by the automaton $A_{M,\ \neg\varphi}$ via a nonemptiness check. If an accepting trace is found, it is returned as a counterexample. If no such trace exists (i.e. the language $\mathscr{L}(A_{M,\ \neg\varphi}) = \emptyset$), we have proved that $M,\ s \models \varphi$. This process is summarized in Table 1.1.

There are two types of model checkers; both use the automata-theoretic approach but

in different ways. Explicit-state model checkers translate specifications to automata explicitly and then use an explicit graph-search algorithm [41]. Symbolic model checkers construct symbolic encodings of these automata and then use a symbolic nonemptiness test [5]. The symbolic construction of the automaton is easier than explicit construction, but the nonemptiness test is computationally demanding.

1.3   Linear Temporal Logic

The focus of this dissertation is on satisfiability checking and model checking using Linear Temporal Logic (LTL) specifications. LTL was first introduced as a vehicle for reasoning about concurrent programs by Pnueli in 1977 [42]. LTL intuitively describes the values of Boolean system variables over linear timelines. The set of LTL operators combines the standard Boolean connectives $\{\neg, \wedge, \vee, \rightarrow\}$, with temporal connectives describing when some event p happens *next time* ($\mathcal{X}p$), *always* ($\Box p$), *eventually* ($\Diamond p$), *until* another event q ($p\mathcal{U}q$), or such that *p releases q* ($p\mathcal{R}q$). LTL is formally defined in Chapter 2.2.1. Example LTL properties appear in Figure 1.2.

LTL model checkers follow the automata-theoretic approach; the complemented LTL specification $\neg\varphi$ is translated to a Büchi automaton[1] $A_{\neg\varphi}$, which is a finite automaton on infinite words that accepts exactly all computations that satisfy the formula $\neg\varphi$.

LTL model checking requires time exponential in the size of the LTL formula. Specifically, let $|M|$ indicate the size of the system model in terms of state space and $|\varphi|$ indicate the size of the specification calculated as the total number of symbols: propositions, logical connectives, and temporal operators. Then the model-checking algorithm for LTL runs in time $|M| \cdot 2^{O(|\varphi|)}$ [43]. The computational complexity of translating the specification $\neg\varphi$ into the Büchi automaton $A_{\neg\varphi}$ is solely responsible for the exponential model checking

[1]Büchi automata are formally defined in Chapter 2.4.

- *Liveness: "Every request is followed by a grant"*
  $\Box(request \rightarrow \Diamond grant)$

- *Invariance: "At some point, p will hold forever"*
  $\Diamond \Box p$

- *"p oscillates every time step"*
  $\Box((p \wedge \mathcal{X} \neg p) \vee (\neg p \wedge \mathcal{X} p))$

- *Safety: "p never happens"*
  $\Box \neg p$

- *Fairness: "p happens infinitely often"*
  $(\Box \Diamond p) \rightarrow \varphi$

- *Mutual exclusion: "Two processes cannot enter their critical sections at the same time"*
  $\Box \neg (in\_CS_1 \wedge in\_CS_2)$

- *Partial correctness: "If p is true initially, then q will be true when the task is completed"*
  $p \rightarrow \Box(done \rightarrow q)$

Figure 1.2 : Examples of LTL properties.

computational complexity for an LTL formula φ. The logic-to-automaton translation step
is clearly a bottleneck in the model-checking algorithm; the problem of checking a Büchi
automaton $A_{M,\,\neg\varphi}$ for nonemptiness is NLOGSPACE-complete [44] and decidable in linear
time [45]. The model-checking problem for LTL has been proved to be PSPACE-complete
[46]. The best algorithms for LTL model checking, which are exponential in the length of
the formula but linear in the size of the model, were proposed by Lichtenstein and Pnueli
[43] and Vardi and Wolper [39], the latter algorithm being the basis for most LTL model-
checking tools today. Thus, developing efficient LTL-to-automaton translation algorithms
is essential for formal verification of modern safety-critical systems.

1.4 Satisfiability Checking for Property Assurance

Inherent to the use of model checking in verification is the writing of formal specifications. Formal behavioral specifications written early in the system-design process and communicated across all design phases have been shown to increase the efficiency, consistency, and quality of the system under development [11, 12]. In *property-based design* the process of enumerating the requirements of the system under design is followed by the formalization of those properties into a precise mathematical logic, such as LTL. Property-based design and other design-for-verification techniques capture design intent precisely, and use formal logic properties both to guide the design process and to integrate verification into the design process [10]. Since there are likely to be errors in the initial properties, the shift to specifying desired system behavior in terms of formal logic properties gives us an opportunity to identify and address these errors in this very initial phase of system design via *property assurance* [12, 47].

The need for checking for errors in formal LTL properties expressing desired system behaviors first arose in the context of model checking. Accordingly, *vacuity checking* aims at reducing the likelihood that a property that is satisfied by the model under verification is an erroneous property [48, 49]. Property assurance is more challenging at the initial phases of property-based design, before a model of the implementation has been specified. *Inherent vacuity checking* is a set of sanity checks that can be applied to a set of temporal properties, even before a model of the system has been developed, but many possible errors cannot be detected by inherent vacuity checking [50].

A stronger sanity check for a set of temporal properties is LTL *realizability* checking, in which we test whether there is an *open system* that satisfies all the properties in the set [51], but such a test is very expensive computationally. In LTL *satisfiability* checking, we test whether there is a *closed system* that satisfies all the properties in the set. We

consider a system open if the system variables can change via external interactions with the environment in which the system is running, whereas a closed system is one in which the environment cannot modify any values of system variables after the initial inputs are provided to the system [52, 53]. The satisfiability test is thus weaker than the realizability test, but its complexity is lower; it has the same complexity as LTL model checking [46], and, as we will show, can be implemented via LTL model checking.

The need for LTL satisfiability checking is widely recognized [54, 55, 56, 57, 58]. Foremost, it serves to ensure that the behavioral description of a system is internally consistent and neither over- or under-constrained. If an LTL property is either *valid*, or *unsatisfiable* this must be due to an error. Consider, for example, the specification *always*($b_1 \rightarrow$ *eventually* b_2), where b_1 and b_2 are propositional formulas. If b_2 is always true, then this property is valid. If b_1 is satisfiable but b_2 is a contradiction, then this property is unsatisfiable. Furthermore, the collective set of properties describing a system must be satisfiable together, to avoid contradictions between different requirements. Satisfiability checking is particularly important when the set of properties describing the design intent continues to evolve, as properties are added and refined, and have to be checked repeatedly. Because of the need to consider large sets of properties, it is critical that the satisfiability test be *scalable*, and able to handle complex temporal properties. This is challenging, as LTL satisfiability, like LTL model checking, is PSPACE-complete [46].

Note that satisfiability checking can be performed via model checking: a *universal model* (that is, a model that allows all possible traces) does not satisfy a linear temporal property $\neg f$ precisely when f is satisfiable. In this thesis, we explore the effectiveness of model checkers as LTL satisfiability checkers. We compare the performance of explicit-state and symbolic model checkers.

1.5   Explicit versus Symbolic Model Checking

LTL model checkers can be classified as *explicit* or *symbolic*. Both types of model checkers create an automaton $A_{M, \neg\varphi}$ such that $\mathscr{L}(A_{M, \neg\varphi}) = \mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$; M satisfies φ iff $\mathscr{L}(A_{M, \neg\varphi}) = \emptyset$. Otherwise, a word accepted by $\mathscr{L}(A_{M, \neg\varphi})$ is returned as a counterexample demonstrating an incorrect computation [40]. The difference between explicit and symbolic model checkers lies in their internal representations of these automata and, subsequently, the algorithms they use to perform the nonemptiness check.

Explicit model checkers, such as Spin[2] [59], construct the state space of the model explicitly. Explicit model checkers search for a trace falsifying the specification via explicit state search, checking one state at a time. The automaton $A_{M, \neg\varphi}$ is not empty if there is a path in the state-transition graph starting in an initial state, reaching some accepting state, and cycling back through that accepting state. The standard algorithm for finding strongly connected components in a state-transition graph is Tarjan's depth-first search algorithm, which runs in time linear in the sum of the number of states and transitions. In practice explicit model checkers usually implement slightly more efficient algorithms to check for nonemptiness by mapping the maximal strongly connected components in the state-transition graph and then checking if any of them contain an accepting state [60, 2, 61]. However, representing and searching the state space explicitly requires a considerable amount of space, even when utilizing optimization techniques such as on-the-fly state space construction [62, 63, 64]. Given that the size of the state space required for model checking is the largest challenge to its efficacy as a verification technique, utilizing techniques that conserve space is vital.

The *state-explosion problem* is widely agreed to be the most formidable challenge facing the application of model checking to large and complex real-world systems [25, 13, 65,

---

[2]http://spinroot.com/

33]. In short, the number of states required to model concurrent systems grows exponentially with the number of system components, constituting the main practical limitation of model checking. Sequential hardware circuits with n input variables and k registers require 2^{n+k} states to represent all possible system configurations [33]. Even simple systems, like an n-bit binary counter, can necessitate large state spaces (in this case, 2^n states). In general, a system with n variables over a domain of k possible values requires at least k^n states in the model. Unfortunately, the state-explosion problem is unavoidable in the worst case. However, a host of techniques have been developed over the last three decades that have successfully eased the problem for certain types of systems. For example, sophisticated data structures, clever algorithms for representing interleaving of concurrent components (called partial order reduction [34]), and the use of bisimulation equivalences [35] and compositional (also called modular) verification [36] to reason about different levels of abstraction, all address the state-explosion problem.

In order to mitigate the state-explosion problem, symbolic model checkers, such as CadenceSMV [14], NuSMV [20], and VIS [66], represent the system model symbolically using sets of states and sets of transitions. They then analyze the state space symbolically using binary decision diagrams (BDDs) [67]. In contrast with explicit-state model checking, states in symbolic model checking, are represented *implicitly*, as a solution to a logical equation. This saves space in memory since syntactically small equations can represent comparatively large sets of states. All symbolic model checkers use the symbolic translation for LTL specifications described in [68] and the analysis algorithm of [69]. The technique of using BDDs in symbolic model checking to reason about Boolean formulas representing the state space, thereby avoiding building the state graph explicitly, was invented by McMillan [13] and is considered to be one of the biggest breakthroughs in the

history of model checking for its impact on the state-explosion problem [70].[3]

1.6 Results

1.6.1 Challenges

There exists a need for algorithms to translate LTL specifications into explicit and symbolic automata in order to enable more efficient specification debugging and model checking to allow scaling to accommodate the size and complexity of modern verification problems.

To achieve this goal we must:

1. Provide an easy-to-use method for sanity checking LTL specifications before model checking in order to increase the likelihood that a mismatch between the specifications and the system model under verification is due to an error in the system model and not an error in the specifications.

2. Develop an understanding of the limitations affecting the efficiency, scalability, and correctness of the current generation of both explicit and symbolic algorithms for LTL-to-automaton translation.

3. Establish a set of representative benchmarks that will be employed to show empirically the efficiency, scalability, and correctness of algorithms for LTL-to-automaton translation, both for LTL satisfiability checking and more general forms of model checking.

4. Understand the complex relationship between the characteristics of an automaton encoding an LTL formula and the performance of the algorithms used to analyze those automata for verification.

---

[3]Others independently published ideas similar to McMillan's symbolic model-checking algorithm at around the same time. See [71] for a high level overview of these techniques.

5. Find new ways to encode LTL formulas as both explicit and symbolic automata in order to improve the performance of analysis by model-checker back-ends.

6. Conclusively illustrate the performance of the developed methods and any significant improvements over the state of the art utilizing the established benchmarks.

7. Propose methods for improving the performance of LTL satisfiability and model checking, taking advantage of modern parallel architectures.

Our contributions address these challenges by describing advances in the domains listed below.

1.6.2 Contributions

The contributions of this dissertation are as follows:

- **[For Challenge 1]** A new default procedure for debugging LTL specifications: LTL satisfiability checking as a front-end to the model-checking process. We show that LTL satisfiability checking is a special case of LTL model checking and advocate the adaptation of this sanity check as a first step for any verification task employing LTL specifications, whether it is debugging specifications for property-based design or checking that a logical system follows a set of specifications via model checking.

- **[For Challenges 2-3]** A varied set of challenging benchmarks that enable objective comparison of LTL translation algorithms, in two parts:

  - A set of benchmarks that we show can provide a basis for fair evaluation of the time-efficiency, correctness, and scalability of algorithms for translating LTL formulas into automata in the context of LTL satisfiability checking. These

benchmarks have recently become the de facto standard for evaluating LTL-to-automaton algorithms in the field.

– A set of benchmarks for LTL model checking of safety properties that serve to objectively evaluate the performance of automata representing LTL specifications in the context of verification via model checking.

- **[For Challenges 2,4]** A comprehensive analysis of all tools that were publicly available when we conducted our study [56] for explicit and symbolic LTL-to-automaton translation in the context of LTL satisfiability checking. We objectively evaluate these algorithms in terms of time-efficiency, correctness, and scalability, whereas previous studies all focused on automaton size. We disprove the popular theory that the performance of such automata is correlated with the automata size, either in terms of number of states, number of transitions, or a combination of both and demonstrate that the symbolic approach is faster than the explicit approach for LTL satisfiability checking.

- **[For Challenges 4-7]** Defining, and proving correct, an extensible set of 29 novel encodings for LTL formulas as symbolic automata, consisting of combinations of different formula normal forms, automaton forms, transition forms, and BDD variable orders. We combine these to form a new multi-encoding approach to symbolic LTL satisfiability checking that can consistently significantly dominate the native translations of the previous state-of-the-art tools for this task, performing up to exponentially better than these tools.

- **[For Challenges 4-7]** Defining, and proving correct, a set of 26 novel explicit automaton encodings of LTL safety formulas in the form of Promela (PROcess MEta LAnguage) `never` claims. These encodings improve the performance of model

checking by utilizing the advantages of varying the state minimization, alphabet representation, state representation, transition encoding, automaton forms, and automaton acceptance conditions. We provide experimental results to show the use of our methods for constructing explicit automata from LTL formulas, and the previous state-of-the-art methods, on a wide set of benchmarks, demonstrating the significant performance gain provided by our methods for LTL model checking.

1.7   Organization of the Dissertation

Chapter 2 introduces the theory underlying the technique of LTL model checking. In Chapter 3 we establish our broad suite of LTL formula benchmarks that has been adopted widely and used to objectively empirically evaluate LTL-to-automaton tools. Next, in Chapter 4, we establish the specification-debugging technique of LTL satisfiability checking and evaluate the state-of-the-art tools for implementing this check utilizing the benchmarks of Chapter 3. Following on the results from Chapter 4, Chapter 5 presents our new multi-encoding approach for LTL symbolic satisfiability checking. Then Chapter 6 describes our improved algorithm for explicit LTL satisfiability checking and model checking. Finally, Chapter 7 concludes with a discussion of our work, its impact, and directions for the future.

Chapter 2

Linear Temporal Logic Model Checking Theory

2.1 Modeling the System

While it is sometimes possible to perform verification directly on a completed system, this approach is undesirable for two reasons. Firstly, it is significantly more efficient and cost-effective to perform verification as early as possible in the system design process, thereby avoiding the possible discovery of an error in the completed system that requires a redesign. Secondly, it is simpler and easier to reason about a model of the system than the system itself because the model includes only the relevant features of the larger system and because the model is easier to build and redesign as necessary.

Models can be extracted from automata, code, scripts or other higher-level specification language descriptions, sets of Boolean formulas, or other mathematical descriptions. They can be comprised of sets of models, such as a set of models of the whole system at different levels of detail, or a set of models of different independent subsystems plus a model of the communications protocol between subsystems, etc. Furthermore, there are many strategies for modeling to optimize clarity, provide generality for reusability, or minimize model checking time, either by reducing time or space complexity during the model-checking step. Models can be designed not only to find bugs in a system design but also to solve problems from other domains. For example, model checking is sometimes used for path planning where the model and the specification property describe the environment to be traversed and the constraints on the path, and the counterexample returned constitutes a

viable path matching those criteria [72].

Whatever the original form of the system model, we eventually translate it into some form of state-transition system; in this thesis we use Büchi automata. The system is described by a set Σ of system variables, also called the system's alphabet. A state is labeled by a set of assignments to the system variables. Each unique variable assignment constitutes a unique state. Since not all variable assignments may be possible, the lower bound on the number of states in a system model is 1 and the upper bound is $2^{|\Sigma|}$, presuming Σ is a set of Boolean variables. We consider a system to transition from one state to another when the values of one or more system variables change from the values they had in the originating state to the values they have in the destination state.

Given that system design and modeling is largely an art form, we simply list here the chief concerns to keep in mind when creating the system model:

- **Defining Σ:** What is the minimum set of system variables required to accurately describe the full set of behaviors of the system?

- **Level of abstraction:** Which details of the system are relevant to the verification process and how do we ensure we have included all of these? Which details serve chiefly to obscure the former? For example, when modeling an automated air traffic control architecture, we include in the model that the controller can request a specific trajectory. However, details of the graphical user interface (GUI) used to make this request are left out.

- **Validation:**

  - **system** $\rightarrow$ **model** Have we modeled the system correctly?

  - **model** $\rightarrow$ **system** How will we ensure the resulting hardware or software system created after verification matches the model we have verified?

2.1.1 Modeling Limitations

Not all systems can be modeled in such a way as to undergo formal verification via symbolic model checking. However the model is created, there is always the chance of creating more states than can be reasoned about in computer memory and having the model-checking step halted by the state explosion problem. This problem can be mitigated during the modeling phase by being cognizant that, in the worst case, the model checker will have to explore the entire state space. Keeping the state space as small as possible by modeling only relevant aspects of the system, minimizing the set of state variables, utilizing sophisticated data structures and abstractions [73], and representing concurrent threads independently to take maximum advantage of partial order reductions performed by the model checker, are some strategies for achieving this goal. (Partial order reduction is a technique that recognizes cases where multiple different interleavings of concurrent processes in the system M have the exact same effect on the property φ, thus only one such sequence needs to be checked [74, 34].)

In this thesis, we presume the system model M to have a finite state space defined over discretely-valued variables. Indeed, the termination of this algorithm is guaranteed by the finite nature of the state space we are exploring [75]. There are many model checkers that capably analyze systems with potentially infinite run-times over finite state spaces. These include Spin [59], NuSMV [20], CadenceSMV [14], and SAL-SMC [76].

Some systems inherently have an infinite state space and thus cannot be model checked using the techniques presented here; instead, these systems must be verified using specialized techniques for providing weaker assurances about larger state spaces or using alternative verification techniques, such as theorem proving. *Bounded model checking (BMC)* [37] can be used to reason about some models whose state spaces exceed the capacity of symbolic model checking. For a given k, BMC tools search for a counterexample of length

k or shorter, so they can prove the presence of errors in the model but cannot be used to prove the absence of errors. Real-time systems with continuous-valued clocks that can be described in terms of linear constraints on clock values can be analyzed using timed automata symbolic model checkers such as UPPAAL [77]. Hybrid automata model checkers like HyTech can verify some types of hybrid systems, that is, systems with a mix of discrete and continuous variables [78] but the infinite nature of the state space means termination of the model-checking algorithm is not guaranteed.

2.2   Specifying the Behavior Property: Linear Temporal Logic

Real-world systems are routinely developed from English-language specifications. Yet, once these systems are built, there is no way to verify that the resulting systems follow the English specifications. Consequently, there is also no way to tell whether following the natural language specifications is a desirable goal, i.e. whether the set of specifications is internally consistent, logically sound, or complete. For formal verification, English is quite simply too imprecise. Consider the statement by Groucho Marx, "I once shot an elephant in my pajamas. How he got in my pajamas I'll never know," or the double-meaning of "Students hate annoying professors." In order to check that a system models its specifications, it is necessary to have a formal, and very precise, notion of exactly what the specifications say. Therefore, we use logic for the specification language. Logic has the advantage of being able to express system specifications in a concise and unambiguous way that is mathematically rigorous and thereby enables automation of the verification process.

We formalize simple grammatical connections using the operators of *propositional logic*: ¬ (negation), ∧ (conjunction), ∨ (disjunction), → (implication); see Table 2.1. We reason about *atomic propositions*, which are Boolean-valued variables. We refer to the set

Propositional Logic:	
$\neg p$	not
$p \wedge q$	and
$p \vee q$	or
$p \rightarrow q$	implies

Table 2.1 : List of the operators of propositional logic.

of atomic propositions as *Prop*. A formula is either an atomic proposition or a sentence comprised of atomic propositions connected by logical operators. The truth value of a formula is determined by an assignment $\mathscr{A}$, which is a mapping of atomic propositions in the domain Σ to truth values: $\mathscr{A} : \Sigma \rightarrow \{0, 1\}$. Thus if there are n atomic propositions in the domain, there are 2^n possible assignments for a given formula. A formula φ is satisfied by an assignment $\mathscr{A}$ that causes the overall formula to evaluate to *true*.

2.2.1 Temporal Logics

Continuous systems necessarily involve a notion of time. Propositional logic is not expressive enough to describe such real systems. Yet, English descriptions are even less precise once we involve time. Consider, for example, the following English sentences:

- I said I would see you on Tuesday.

- "This is the worst disaster in California since I was elected." [California Governor Pat Brown]

- "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone." [Kansas State Legislature, early 1890's]

As a result, Amir Pnueli introduced the notion of using temporal logics, which were originally developed by philosophers for investigating how time is used in natural lan-

guage arguments, to reason about concurrent systems [42]. (Burstall [79] and Kröger [80] independently proposed the use of weaker forms of temporal reasoning about computer programs at roughly the same time.) Temporal logics are modal logics geared toward the description of the temporal ordering of events. For most safety-critical systems, tying the system model to an explicit universal clock is overkill; it exacerbates the state explosion problem without adding value to the verification process. It is sufficient simply to guarantee that events happen in a certain partial (or, in some cases, total) order. Temporal logics were invented for this purpose. Because they describe the ordering of events in time without introducing time explicitly, temporal logics are particularly effective for describing concurrent systems [81]. Temporal logics conceptualize time in one of two ways. Linear temporal logics consider every moment in time as having a unique possible future. Essentially, they reason over a classical timeline. In branching temporal logics, each moment in time may split into several possible futures. In essence, these logics view the structure of time as a tree, rooted at the current time, with any number of branching paths from each node of the tree.

Pnueli's historic 1977 paper [42] proposed the logic LTL. This logic extended propositional logic with temporal operators $\mathcal{G}$ (for "globally") and $\mathcal{F}$ (for "in the future"), which we also call □ and ◇, respectively, and introduced the concept of *fairness*, which ensures an infinite-paths semantics. LTL was subsequently extended to include the $\mathcal{U}$ ("until") operator originally introduced by Kamp in 1968 [82] and the $\mathcal{X}$ ("next time") operator from the temporal logic $\mathcal{U}\mathcal{B}$ (the *unified* system of *branching* time) [83]. In 1981, Clarke and Emerson extended $\mathcal{U}\mathcal{B}$, thereby inventing the branching temporal logic they named Computational Tree Logic (CTL) and, with it, CTL model checking. Lichtenstein and Pnueli defined model checking for LTL in 1985 [43]. (For a more detailed history of the technical developments that lead from Prior's early observations on time and Church's logical spec-

ification of sequential circuits to LTL and automata-theoretic model checking, see [84].) The combination of LTL and CTL, called CTL*, was defined by Emerson and Halpern in 1986, though there are currently no industrial model checkers for this language.[1] Since both explicit and symbolic model checking were originally defined for CTL, and since LTL model checking is frequently defined in the literature in relation to CTL model checking, we would be remiss not to mention the logic CTL here. Therefore, we follow the formal definition of LTL below with a brief description of the logics CTL and CTL* along with a short discussion of the merits and expressibility of LTL that motivate our focus on LTL satisfiability checking and LTL model checking.

LTL

Linear Temporal Logic (LTL) reasons over linear traces through time. At each time instant, there is only one real future timeline that will occur. Traditionally, that timeline is defined as starting "now," in the current time step, and progressing infinitely into the future.

LTL formulas are composed of a finite set *Prop* of atomic propositions, the Boolean connectives ¬, ∧, ∨, and →, and the temporal connectives $\mathcal{U}$ (until), $\mathcal{R}$ (release), $\mathcal{X}$ (also called ○ for "next time"), □ (also called $\mathcal{G}$ for "globally"), and ◇ (also called $\mathcal{F}$ for "in the future"). Intuitively, $\varphi \, \mathcal{U} \, \psi$ states that either ψ is true now or φ is true now and φ remains true until such a time when ψ holds. Dually, $\varphi \, \mathcal{R} \, \psi$, stated φ *releases* ψ, signifies that ψ must be true now and remain true until such a time when φ is true, thus releasing ψ. $\mathcal{X}\varphi$ means that φ is true in the next time step after the current one. Finally, □φ commits φ to being true in every time step while ◇φ designates that φ must either be true now or at some future time step. We define LTL formulas inductively:

---

[1]There was one research prototype CTL* model checker called AltMC (**Alt**ernating Automata-based **M**odel **C**hecker). Visser and Barringer created AltMC in 1998 and explained how it could be integrated into the industrial model checker Spin [85].

Definition 1

For every $p \in Prop$, p is a formula. If φ and ψ are formulas, then so are:

$$\neg\varphi \qquad \varphi \wedge \psi \qquad \varphi \rightarrow \psi \qquad \varphi \, \mathcal{U} \, \psi \qquad \square\varphi$$

$$\varphi \vee \psi \qquad \mathcal{X}\varphi \qquad \varphi \, \mathcal{R} \, \psi \qquad \diamond\varphi$$

Furthermore, we define the closure of LTL formula φ, $cl(\varphi)$, as the set of all of the subformulas of φ and their negations (with redundancies, such as φ and $\neg\neg\varphi$, consolidated). LTL formulas describe the behavior of the variables in *Prop* over a linear series of time steps starting at time zero and extending infinitely into the future.[2] We satisfy such formulas over *computations*, which are functions that assign truth values to the elements of *Prop* at each time instant [87]. In essence, a computation path π satisfies a temporal formula φ if φ is true in the zeroth time step of π, π_0.

Definition 2

We interpret LTL formulas over computations of the form $\pi : \omega \rightarrow 2^{Prop}$, where ω is used in the standard way to denote the set of non-negative integers. We also use iff to abbreviate "if and only if." We define $\pi, i \vDash \varphi$ (computation π at time instant $i \in \omega$ satisfies LTL formula φ) as follows:

- $\pi, i \vDash p$ for $p \in Prop$ iff $p \in \pi(i)$.

- $\pi, i \vDash \neg\varphi$ iff $\pi, i \nvDash \varphi$.

- $\pi, i \vDash \varphi \wedge \psi$ iff $\pi, i \vDash \varphi$ and $\pi, i \vDash \psi$.

- $\pi, i \vDash \varphi \vee \psi$ iff $\pi, i \vDash \varphi$ or $\pi, i \vDash \psi$.

- $\pi, i \vDash \mathcal{X}\varphi$ iff $\pi, i + 1 \vDash \varphi$.

---

[2]Though we will not discuss them here, some variations of LTL also consider time steps that have happened in the past. For example, we could add past-time versions of $\mathcal{X}$ and $\mathcal{U}$ called $\mathcal{Y}$ (also called $\ominus$ for "previous time" or "yesterday") and $\mathcal{S}$ (since), respectively. Past-time LTL was first introduced by Kamp in 1968 [82] but does not add expressive power to the future-time LTL defined here [86].

- $\pi, i \vDash \varphi \, \mathcal{U} \, \psi$ iff $\exists j \geq i$, such that $\pi, j \vDash \psi$ and $\forall k, i \leq k < j$, we have $\pi, k \vDash \varphi$.

- $\pi, i \vDash \varphi \, \mathcal{R} \, \psi$ iff $\forall j \geq i$, iff $\pi, j \nvDash \psi$, then $\exists k, i \leq k < j$, such that $\pi, k \vDash \varphi$.

- $\pi, i \vDash \square \varphi$ iff $\forall j \geq i, \pi, j \vDash \varphi$.

- $\pi, i \vDash \Diamond \varphi$ iff $\exists j \geq i$, such that $\pi, j \vDash \varphi$.

We take $\vDash (\varphi)$ to be the set of computations that satisfy φ at time 0, i.e., $\{\pi : \pi, 0 \vDash \varphi\}$. We define the prefix of an infinite computation π to be the finite sequence starting from the zeroth time step, $\pi_0, \pi_1, \ldots, \pi_i$ for some $i \geq 0$.

Equivalences   While we have presented the most common LTL syntax, operator equivalences allow us to reason about LTL using a reduced set of operators. In particular, the most common minimum set of LTL operators is $\neg$, $\vee$, $\mathcal{X}$, and $\mathcal{U}$. From propositional logic, we know that $(\varphi \rightarrow \psi)$ is equivalent to $(\neg \varphi \vee \psi)$ by definition and that $(\varphi \wedge \psi)$ is equivalent to $\neg(\neg \varphi \vee \neg \psi)$ by DeMorgan's law. We can define $(\Diamond \varphi)$ as $(\textit{true} \, \mathcal{U} \varphi)$. (Similarly, $(\square \varphi) \equiv (\textit{false} \, \mathcal{R} \, \varphi)$.) The *expansion laws* state that $(\varphi \, \mathcal{U} \, \psi) = \psi \vee [\varphi \wedge \mathcal{X}(\varphi \, \mathcal{U} \, \psi)]$ and $(\varphi \, \mathcal{R} \, \psi) = \psi \wedge [\varphi \vee \mathcal{X}(\varphi \, \mathcal{R} \, \psi)]$. The operators $\square$ and $\Diamond$ are logical duals as $(\square \varphi)$ is equivalent to $(\neg \Diamond \neg \varphi)$ and $(\Diamond \varphi)$ is equivalent to $(\neg \square \neg \varphi)$. Finally, $\mathcal{U}$ and $\mathcal{R}$ are also logical duals. Since this last relationship is not intuitive, we offer proof below. (Incidentally, $\mathcal{X}$ is the dual of itself: $(\neg \mathcal{X} \varphi) \equiv (\mathcal{X} \neg \varphi)$.)

Informally, $\varphi \, \mathcal{U} \, \psi$ signifies that either ψ is true now (in the current time step) or φ is true now and for every following time step until such a time when ψ is true. Note that this operator is also referred to as *strong until* because ψ *must* be true at some time. Conversely, *weak until*, sometimes included as the operator $\mathcal{W}$, is also satisfied if φ is true continuously but ψ is never true. Formally,

$$\pi, i \vDash \varphi \, \mathcal{W} \, \psi \text{ iff either } \forall j \geq i, \pi, j \vDash \varphi \text{ or } \exists j \geq i, \text{ such that } \pi, j \vDash \psi \text{ and } \forall k, i \leq k < j,$$

$$\text{we have } \pi, k \vDash \varphi.$$

$\mathcal{R}$ is the dual of $\mathcal{U}$, so $\varphi \, \mathcal{R} \, \psi = \neg(\neg\varphi \, \mathcal{U} \, \neg\psi)$. We say that "$\varphi$ releases ψ" because ψ must be true now and, in the future, $\neg\psi$ implies φ was previously true. Note that at that point in the past, both φ and ψ were true at the same time and that φ may never be true, as long as ψ is always true. We can define $\mathcal{R}$ in terms of the $\mathcal{U}$-operator using the following lemma, where iff is used in the standard way to abbreviate "if and only if."

Lemma 2.1

$\neg(\neg a \, \mathcal{U} \, \neg b) = a \, \mathcal{R} \, b.$

Proof: We use the formal semantic definitions of $\mathcal{U}$ and $\mathcal{R}$, given in [88].

$$\pi, i \models \xi \, \mathcal{U} \, \psi \quad \text{iff} \quad \text{for some } j \geq i, \text{ we have } \pi, j \models \psi$$

$$\text{and for all } k, \ i \leq k < j, \text{ we have } \pi, k \models \xi.$$

$$\pi, i \models \neg(\xi) \, \mathcal{U} \, \neg(\psi) \quad \text{iff} \quad \text{for some } j \geq i, \text{ we have } \pi, j \models \neg(\psi)$$

$$\text{and for all } k, \ i \leq k < j, \text{ we have } \pi, k \models \neg(\xi).$$

$$\pi, i \models \neg(\xi) \, \mathcal{U} \, \neg(\psi) \quad \text{iff} \quad ((\exists \, j \geq i : \pi, j \models \neg(\psi))$$

$$\wedge (\forall \, k, \ i \leq k < j : \pi, k \models \neg(\xi))).$$

$$\pi, i \models \neg(\neg(\xi) \, \mathcal{U} \, \neg(\psi)) \quad \text{iff} \quad \neg((\exists \, j \geq i : \pi, j \models \neg(\psi))$$

$$\wedge (\forall \, k, \ i \leq k < j : \pi, k \models \neg(\xi))).$$

$$\text{iff} \quad (\neg(\exists \, j \geq i : \pi, j \models \neg(\psi))$$

$$\vee \neg(\forall \, k, \ i \leq k < j : \pi, k \models \neg(\xi))).$$

$$\text{iff} \quad ((\forall \, j \geq i : \pi, j \not\models \neg(\psi))$$

$$\vee (\exists \, k, \ i \leq k < j : \pi, k \not\models \neg(\xi))).$$

$$\text{iff} \quad (\forall \, j \geq i : \; \pi, j \models \psi$$

$$\vee \exists \, k, \; i \leq k < j : \; \pi, k \models \xi).$$

$$\text{iff} \quad (\forall \, j \geq i : \; \pi, j \not\models \psi$$

$$\rightarrow \exists \, k, \; i \leq k < j : \; \pi, k \models \xi).$$

$$\text{iff} \quad \text{for all } j \geq i \text{ if } \pi, j \not\models \psi,$$

$$\text{then for some } k, \; i \leq k < j \text{ we have } \pi, k \models \xi.$$

$$\text{iff} \quad \pi, i \models \xi \, \mathcal{R} \, \psi. \quad \blacksquare$$

2.2.2   Logical Expressiveness of LTL

Inevitably the question arises: why LTL? What are the alternative specification logics? Why (and when) should we choose LTL? In order to put LTL in the proper context, we briefly discuss the most viable alternatives and discuss when LTL is (and is not) an appropriate choice.

CTL

Computational Tree Logic (CTL) is a branching time logic that reasons over many possible traces through time. Historically, CTL was the first logic used in model checking [6] and it remains a popular specification logic. Unlike LTL, for which every time instance has exactly one immediate successor, in CTL a time instance has a finite, non-zero number of immediate successors. A branching timeline starts in the current time step, and may progress to any one of potentially many possible infinite futures. In addition to reasoning along a timeline, as we did for linear time logic, branching time temporal operators must also reason across the possible branches. Consequently, the temporal operators in CTL are all two-part operators with one part specifying, similarly to LTL, the action to occur along a future timeline and another part specifying whether this action takes place on at least one

---

Linear Temporal Logic (LTL) formulas reason about linear timelines:

- a finite set *Prop* of atomic propositions

- Boolean connectives: ¬, ∧, ∨, and →

- temporal connectives:
  $\mathcal{X}\varphi$ next time
  $\varphi\,\mathcal{U}\,\psi$ until
  $\varphi\,\mathcal{R}\,\psi$ release
  $\Box\varphi$ also called $\mathcal{G}$ for "globally"
  $\Diamond\varphi$ also called $\mathcal{F}$ for "in the future"

Computational Tree Logic (CTL) reasons about branching paths:

- temporal connectives are always proceeded by path quantifiers:
  $\mathcal{A}$ for all paths
  $\mathcal{E}$ exists a path

Figure 2.1 : Syntax of LTL and CTL.

branch or all branches.

There is an automata-theoretic approach to branching time model checking, similar to the ideas described in this dissertation for LTL. While we do not translate branching temporal formulas into nondeterministic tree automata due to the double exponential blow-up inherent in this process, we can translate them into alternating tree automata in linear time [89].

LTL vs CTL

The logical expressiveness of LTL and CTL is incomparable [90]. Since CTL allows explicit existential quantification over paths, it is more expressive in some cases where we want to reason about the possibility of the existence of a specific path through the transition

system model M, such as when M is best described as a computation tree. For example, there are no LTL equivalents of the CTL formulas ($\mathcal{E}\mathcal{X}\ p$) and ($\mathcal{A}\square\mathcal{E}\diamond p$) since LTL cannot express the possibility of p happening on some path (but not necessarily all paths) next time, or in the future. LTL describes executions of the system, not the way in which they can be organized into a branching tree. Intuitively, it is difficult (or impossible) to express in LTL situations where distinct behaviors occur on distinct branches at the same time. Conversely, it is difficult (or impossible) to express in CTL

Expressiveness

Figure 2.2 : Venn diagram showing the expressiveness of common temporal logics.

some situations where the same behavior may occur on distinct branches at distinct times, a situation where the ability of LTL to describe individual paths is quite useful. Realistically, the former rarely happens and LTL turns out to be more expressive from a practical point of view than CTL [91].

Model Checking Time Complexity   The model-checking algorithms for LTL and CTL are different. The major argument in favor of using a branching time logic, like CTL, instead of LTL for property specification is that the model-checking problem for CTL has lower computational complexity than for LTL. Specifically, let $|M|$ indicate the size of the system model in terms of state space and $|\varphi|$ indicate the size of the specification calculated as the total number of symbols: propositions, logical connectives, and temporal operators. Then the model-checking algorithm for CTL runs in time $O(|M||\varphi|)$ [92] and the model-checking algorithm for LTL runs in time $|M| \cdot 2^{O(|\varphi|)}$ [43]. Intuitively, this is because CTL is state-based (i.e. reasoning over states in time), and this set of states is easily converted into an automaton whereas the succinct path-based model of LTL (where many possible paths may pass through a single state) must be expanded. This difference in the computational

complexity of translating the specification $\neg\varphi$ into the Büchi automaton $A_{\neg\varphi}$ is solely responsible for the difference in the model checking computational complexity for an LTL formula φ and for a CTL formula φ. There is no hope of reconciling the time complexity of the general model-checking problem for LTL with that of CTL since the model-checking problem for LTL is PSPACE-complete [46].

However, the comparison of the specification logics LTL and CTL by time complexity is somewhat misleading and certainly not significant enough to be the sole deciding factor of which logic to use for specification. Though model checking can be done in time linear in the size of the specification for CTL [92, 7] the requirement for double-operators means that CTL formulas tend to be longer and more complicated than LTL formulas. In fact, it is not entirely clear how much effect the exponential blow-up for LTL model checking has in practice given that the size of most LTL specifications is very small. Furthermore, if we examine specific, practical variants of the model-checking problem for CTL, we find that the complexity dominance of this logic does not hold [91]. Performing specification debugging via satisfiability checking as we describe in this dissertation is still PSPACE-complete for LTL [46] but is EXPTIME-complete for CTL [93, 94]. For the practical model checking applications of compositional verification, verification of open or reactive systems (i.e. those systems that interact with an environment), verification of concurrent systems, and automata-theoretic verification, LTL-based algorithms either dominate those for CTL or perform similarly [95].

Since there are fast, efficient tools for LTL model checking, and it is questionable to what extent this theoretical difference affects the process of model checking in practice, we focus on the language that is more suitable for specification. From a system design point of view, it is more important to be able to write clear and correct specifications to input into the model checker.

Usually, we want to describe behavioral, rather than structural, properties of the model,

making LTL the better choice for specification since such properties are easily expressed in LTL but may not be expressible in CTL. For example, we may want to say that p happens within the next two time steps, ($\mathcal{X} p \vee \mathcal{X}\mathcal{X} p$), or that if p ever happens, q will happen too, ($\Diamond p \rightarrow \Diamond q$), neither of which is expressible in CTL [96]. Similarly, we cannot state in CTL that if p happens in the future, q will happen in the next time step after that, which is simply $\Diamond(p \wedge \mathcal{X} q)$ in LTL [33]. Worst of all, it is not obvious that these useful properties should not be expressible in CTL. Indeed, a thorough comparison of the two logics concludes that CTL is unintuitive, hard to use, ill-suited for compositional reasoning, and fundamentally incompatible with semi-formal verification while LTL suffers from none of these inherent limitations and is better suited to model checking in general [91].

Fairness LTL is a fair language whereas CTL is not. That is to say that LTL can express properties of fairness, including both *strong fairness* and *weak fairness* whereas CTL cannot (though CTL model checkers generally allow users to specify fairness statements separately to account for this shortcoming). For example, the LTL formula ($\Box\Diamond p \rightarrow \Diamond q$), which expresses the property that infinitely many p's implies eventually q, is the form of a common fairness statement: a continuous request will eventually be acknowledged. Yet this sentiment is not expressible in CTL. Since fairness properties occur frequently in the specifications we wish to verify about real-life reactive systems, this adds to the desirability of LTL as a specification language.

For another example, the common invariance property $\Diamond\Box p$, meaning "at some point, p will hold forever" cannot be expressed in CTL. It is difficult to see why this formula is not equivalent to the CTL formula $\mathcal{A}\Diamond\mathcal{A}\Box p$; after all, we are basically claiming that on all paths, there's a point where p holds in all future states. (The standard semantic interpretation of LTL corresponds to the "for all paths" syntax of CTL. For this reason, we consider there to be an implicit $\mathcal{A}$ operator in front of all LTL formulas when we compare
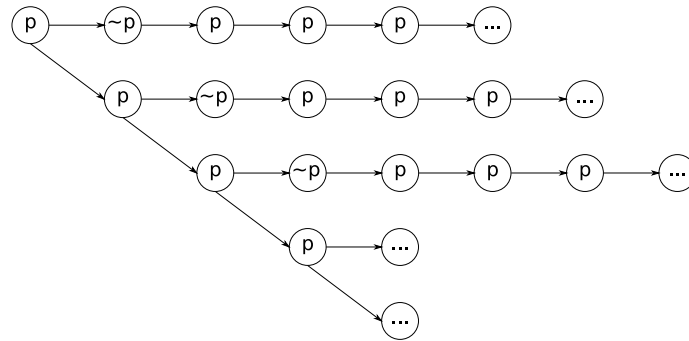
Figure 2.3 : A situation where the LTL formula $\Diamond\Box p$ holds but the CTL formula ($\mathcal{A}\Diamond\mathcal{A}\Box p$) does not.

them to CTL formulas.) To illustrate the inequivalence of these two formulas, we show in Figure 2.3 a timeline that satisfies $\Diamond\Box p$ but does not satisfy $\mathcal{A}\Diamond\mathcal{A}\Box p$. Note that $\mathcal{A}\Diamond\mathcal{A}\Box p$ does not hold along the vertical spine in particular – there is never a point where $\mathcal{A}\Box p$ holds along this path. There is always one child where p holds forever and one child where $\neg p$ holds.
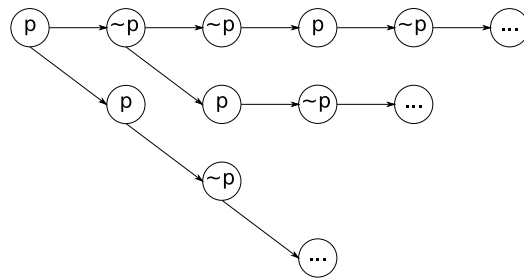
Figure 2.4 : A situation where the three equivalent formulas LTL formula $\mathcal{X}\Diamond p$, LTL formula $\Diamond\mathcal{X}\ p$, and CTL formula $\mathcal{A}\mathcal{X}\mathcal{A}\Diamond\ p$ hold but the CTL formula ($\mathcal{A}\Diamond\mathcal{A}\mathcal{X}\ p$), which is strictly stronger, does not.

Another frequently cited example of the unintuitiveness of CTL is the property that the CTL formula $\mathcal{A}\mathcal{X}\mathcal{A}\Diamond\ p$ is not equivalent to the formula ($\mathcal{A}\Diamond\mathcal{A}\mathcal{X}\ p$). Again, the distinction is subtle. The former formula states that, as of the next time step, it is true that p will definitely hold at some point in the future or, in other words, p holds sometime in the strict

future. This formula is equivalent to the LTL formulas $\mathcal{X}\diamond p$ and $\diamond\mathcal{X}\ p$. On the other hand, the meaning of ($\mathcal{A}\diamond\mathcal{A}\mathcal{X}\ p$) is the strictly stronger (and actually quite strange) assertion that on all paths, in the future, there is some point where p is true in the next time step on all of the branches from that point. Figure 2.4 illustrates this subtlety. Note that in this timeline, p is true in the strict future along every path but there is not a point on every path where p is true in the next step on all branches.

These examples illustrate why LTL is frequently considered a more straightforward language, better suited to specification and more usable for verification engineers and system designers. LTL is the preferred logic of the two for general property specification and on-the-fly verification [97]. Verification engineers have found expressing specifications in LTL to be more intuitive, and less error-prone, than using CTL, particularly for verifying concurrent software architectures [98]. The vast majority of CTL specifications used in practice are equivalent to LTL specifications; it is rare that the added ability to reason over computation tree branches is needed and it frequently requires engineers to look at their designs in an unnatural way [91]. The expressiveness, simplicity, and usability of LTL, particularly for practical applications like open system or compositional verification, and specification debugging, make it a good choice for industrial verification.

CTL*

Emerson and Halpern first invented the logic CTL* in 1983 [99]. This logic combines the syntaxes of the two logics LTL and CTL. It includes all of the logical operators in LTL and both path quantifiers of CTL but does not have the CTL restriction that temporal operators must appear in pairs. Both LTL and CTL are proper subsets of CTL*, as are all combinations of LTL and CTL formulas; CTL* is more expressive than both LTL and CTL combined. For example, the formulas $E(\square\diamond p)$ and $A(\diamond\square p) \vee A\square(E\diamond p)$ are both in CTL* but in neither LTL nor CTL.

However, this expressive power comes at a great cost. The model-checking problem for CTL* is PSPACE-complete [46], which the same general complexity as for LTL, though the algorithm is considerably more complex to implement and there are currently no model checkers for this logic. Indeed, for practical model-checking problems such as compositional verification and verification of open or reactive systems (i.e. those systems that interact with an environment) CTL* is dominated by LTL [95]. Furthermore, the simple task of specification debugging via satisfiability checking is 2EXPTIME-complete for CTL* [100, 101]. Simply translating a CTL* specification into an automaton for model checking involves a doubly-exponential blow-up [102]. So, despite the deceptive time complexity for the general model-checking problem, adding branching to LTL is not free. In practice, should LTL prove to be too limited to express a desired property, CTL* is almost certainly sufficient [97]. However, the lack of industrial model-checking tools that accept CTL* specifications is a deterrent to the use of this logic.

Industrial Logics Based on LTL

In several cases, industrial companies have defined extensions of LTL, specialized for their verification needs. Usually these linear time logics add operators to make the expression of specific properties easier and to extend the specification language to full ω-regularity. The optimal position in the trade-off between logical expressiveness and model checking time complexity remains under discussion.

Definition 3

*An ω-**regular expression** is an expression of the form $\bigcup_i \alpha_i(\beta_i)^\omega$ where i is non-zero and finite and α and β are regular expressions over the alphabet Σ.*

We refer to the standard definition of a regular expression, comprised of the elements of the alphabet Σ, parentheses, and the operators $+$, $\cdot$, and $*$ for union, concatenation, and star-

closure, respectively. An ω-regular expression adds the exponent ω that, in some sense, extends the $*$-exponent since a^* designates an arbitrary, possibly zero, finite number of repetitions of a while a^ω designates an infinite number of repetitions of a.

Definition 4

*An ω-**regular language** is one that can be described by an ω-regular expression. Also, a language is ω-regular if and only if there exists a Büchi automaton that accepts it. (We discuss Büchi automata in detail in Chapter 2.4.) The family of ω-regular languages is closed under union, intersection, and complementation.*

LTL can express a strict subset of ω-regular expressions; it can describe specifically the $*$-free ω-regular events. For example, LTL cannot express the sentiment that a particular event must occur exactly every n time steps of an infinite computation and that this event may or may not occur during any of the other time steps. Wolper first pointed this out and defined Extended Temporal Logic (ETL), which augmented LTL with operators corresponding to right-linear grammars, thus expanding the expressiveness to all properties that can be described by ω-regular expressions [103]. Vardi and Wolper followed this by proposing ETLs where the temporal operators are defined by finite ω-automata, which provide useful tools for hardware specification [44]. However, model checking with ETL involves a difficult complementation construction for Büchi automata [104]. Banieqbal and Barringer and, separately, Vardi created a linear μ-calculus by extending LTL with fixpoint operators, which allows for a more natural description of constructs like recursive procedures and modules in compositional verification [105, 106]. Sistla, Vardi, and Wolper's Quantified Propositional Temporal Logic (QPTL) avoids common user difficulties with fixpoint calculi and achieves ω-regularity instead by allowing quantification over propositional variables but at the cost of a nonelementary time complexity [107]. Emerson and Trefler proposed dealing with real-time correctness properties while avoiding the

nonelementary time complexity by using Real Time Propositional LTL (RTPLTL), which adds time bounds to temporal operators referencing multiple independent clocks but can be checked in time exponential in the size of the regular expression [108].

Verification engineers in industry have also extended LTL to suit their specific needs. After successful verification efforts from 1995 to 1999 using symbolic model checking with specifications in FSL, a home-grown linear temporal logic specialized for hardware checking [109], Intel developed the formal specification language, ForSpec [110]. The temporal logic underlying ForSpec, called FTL, extends past-time LTL to add explicit operators for manipulating multiple clocks and reset signals, expressions for reasoning about regular events, and time bounds on the temporal operators that allow users to specify time windows during which an operator is satisfied. The cost of the increased expressivity of FTL is that checking satisfiability is EXPSPACE-complete. IBM developed their own "syntactic sugar" [111] in parallel from the early 1990s. IBM's Sugar extends LTL with Sugar Extended Regular Expressions (SEREs), the ability to explicitly reference multiple independent clocks, and limited Optional Branching Extensions (OBEs) [112]. Motorola's CBV and Verisity's Temporal e [113] are also linear time logics that achieve full ω-regularity by adding regular expressions and clock operations. In 2003, the standardization committee Accellera, considering these four industrial specification languages, announced the industry standard languages SystemVerilog Assertion (SVA) language [114] and Property Specification Language (PSL), which is based chiefly on Sugar with heavy influence from ForSpec [115, 116]. It is worth noting that restricted variants of LTL have also proved useful for specializing the logic without increasing computational complexity. For example, Allen Linear Temporal Logic (ALTL) [117] marries a restricted LTL, without $\mathcal{X}$-, $\mathcal{U}$-, or $\mathcal{R}$-operators, with Allen's temporal intervals [118], but ALTL satisfiability checking is only NP-complete.

In summary, the industrial languages described in this section are all based on LTL and use model checking methods similar to those described in this dissertation. Thus, insights into LTL model checking extend to all of these languages.

2.3   Specification Debugging via Satisfiability Checking

Just as designing a bug-free system is difficult (motivating us to employ model checking to find the bugs), writing correct specifications to check the system is difficult. Therefore, an important step in between writing specifications and performing the model-checking step is specification debugging. As specifications are usually significantly smaller than the systems they describe, they are significantly easier to check. The goal of specification debugging is to answer, as best as possible, the question "do the specifications say what I meant?" Though it is impossible to answer this question absolutely, it is usually sufficient to run a series of tests on the set of specifications meant to flag situations where the specifications clearly cannot match their authors' intentions.

When the model does not satisfy the specification, model-checking tools accompany this negative answer with a counterexample, which points to an inconsistency between the system and the desired behaviors. It is often the case that there is an error in the system model or in the formal specification. Such errors may not be detected when the answer of the model-checking tool is positive: while a positive answer does guarantee that the model satisfies the specification, the answer to the real question, namely, whether the system has the intended behavior, may be different. We need to ask two questions:

1. If there is disagreement between the system model and the specification, which one has the error?

2. If there is agreement, is this caused by an error? (A positive model checking result does not mean there is no error. For example, the specification could have a bug!)

Testing can be performed by specification authors writing small system models that they believe should or should not satisfy each specification and then verifying this is the case via model checking. However, this process can be time-consuming, tedious, and error-prone.

The realization of this unfortunate situation has led to the development of several *sanity checks* for formal verification [119]. The goal of these checks is to detect errors in the system model or the properties. Sanity checks in industrial tools are typically simple, ad hoc tests, such as checking for enabling conditions that are never enabled [120]. Standard specification testing can also be preformed more intelligently using *concept analysis*, which produces a hierarchical set of clusters of test traces grouped by similarities [121]. This technique allows the specification author to inspect a small number of clusters instead of a large number of individual traces and use the similarities in the clusters to help determine whether the set of traces in each cluster points to specification error(s) or not.

Of course, it is extremely desirable to run automated tests on specifications. *Vacuity detection* provides a systematic approach for checking whether a subformula of the specification does not affect the satisfaction of the specification in the model. Intuitively, a specification is satisfied vacuously in a model if it is satisfied in some non-interesting way. For example, the LTL specification $\Box(req \rightarrow \Diamond grant)$ ("every request is eventually followed by a grant") is satisfied vacuously in a model with no requests. While vacuity checking cannot ensure that whenever a model satisfies a formula the model is correct, it does identify certain positive results as vacuous, increasing the likelihood of capturing modeling and specification errors. Several papers on vacuity checking have been published over the last few years [122, 48, 123, 124, 125, 49, 126, 127], and various industrial model-checking tools support vacuity checking [122, 48, 123].

At a minimum, it is necessary to employ *LTL satisfiability checking* [56]. If the specification is *valid*, that is, true in *all* models, then model checking this specification always results in a positive answer. Basically, the specification is irrelevant. Consider for example

the specification $\Box(b_1 \rightarrow \Diamond b_2)$, where b_1 and b_2 are propositional formulas. If b_1 and b_2 are logically equivalent, then this specification is valid and is satisfied by all models. Clearly, if a formal property is valid, then this is certainly due to an error. Similarly, if a formal property is *unsatisfiable*, that is, true in *no* model, then this is also certainly due to an error. For example, if the specification is $\Box(b_1 \wedge \Diamond b_2)$ where b_1 and b_2 are contradictory, then the specification can never be true. Even if each individual property written by the specifier is satisfiable, their conjunction may very well be unsatisfiable. Recall that a logical formula φ is valid iff its negation $\neg\varphi$ is not satisfiable. Thus, as a necessary sanity check for debugging a specification, we must ensure that both the specification φ and its negation $\neg\varphi$ are satisfiable and that the conjunction of all specifications is satisfiable.

Fortunately, this check is easily performed using standard model-checking tools to check the specifications against a *universal model* before checking them against the system model. A basic observation underlying this conclusion is that LTL satisfiability checking can be reduced to model checking. Consider a formula φ over a set *Prop* of atomic propositions. If a model M is *universal*,[3] that is, it contains all possible traces over *Prop*, then φ is satisfiable precisely when the model M does *not* satisfy $\neg\varphi$. Thus, it is easy to include satisfiability checking using LTL model-checking tools as a specification debugging step in the verification process.[4]

2.4   LTL-to-Automaton

In LTL model checking, we check LTL formulas representing desired behaviors against a formal model of the system designed to exhibit these behaviors. To accomplish this task,

---

[3]See Chapter 3.5 for implementations of universal models.

[4]In contrast, for branching time logics such as CTL or CTL*, satisfiability checking is significantly harder than model checking. For LTL, both satisfiability checking and model checking are PSPACE-complete with respect to formula size [46]. On the other hand, with respect to formula size, model checking is NLOGSPACE-complete for CTL [128] and PSPACE-complete for CTL* [45], while satisfiability is EXPTIME-complete for CTL [94, 93] and 2EXPTIME-complete for CTL* [101, 100]

we employ the automata-theoretic approach, in which the LTL formulas must be translated into Büchi automata[5] [39]. This step is performed automatically by the model checker.

Definition 5

*A **Büchi Automaton** (BA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where:*

- *Q is a finite set of states.*

- *Σ is a finite alphabet.*

- *$\delta : Q \times \Sigma \times Q$ is a transition relation.*

- *$q_0 \in Q$ is the initial state.*

- *$F \subseteq Q$ is a set of final states.*

A run of a Büchi automaton over an infinite word $w = w_0, w_1, w_2, \ldots \in \Sigma$ is a sequence of states $q_0, q_1, q_2, \ldots \in Q$ such that $\forall i \geq 0$, $\delta(q_i, w_i) = q_{i+1}$. An infinite word w is accepted by the automaton if the run over w visits at least one state in F infinitely often. We denote the set of infinite words accepted by an automaton A by $\mathcal{L}_\omega(A)$. A Generalized Büchi Automaton (GBA) is one for which F is a set of acceptance sets such that an infinite word w is accepted by the automaton if the run over w visits at least one state in each set in F infinitely often.

We also define the extended transition relation $\delta^\omega : Q \times \Sigma^\omega \times Q$, which takes a string, rather than a single character, as the second argument and yields the result of reading in that string. Let λ represent the empty string. Then we can define δ^ω recursively. For all $q \in Q, w \in \Sigma^\omega, \sigma \in \Sigma$:

$$\delta^\omega(q, \lambda) = q$$

$$\delta^\omega(q, w\sigma) = \delta(\delta^\omega(q, w), \sigma).$$

---

[5]Büchi automata were introduced by J.R. Büchi in 1962 [129].

A computation satisfying LTL formula φ is an infinite word over the alphabet $\Sigma = 2^{Prop}$, which is accepted by the Büchi automaton A_φ corresponding to φ. For this reason, A_φ is also called a *tester* for φ; it characterizes all computations that satisfy φ. The set of computations of φ comprise the language $\mathscr{L}_\omega(A_\varphi)$. Every LTL formula has an equivalent Büchi automaton. The next theorem relates the expressive power of LTL to that of Büchi automata.

Theorem 2.1

Given an LTL formula φ, we can construct a nondeterministic Büchi automaton $A_\varphi = \langle Q, \Sigma, \delta, q_0, F \rangle$ such that $|Q|$ is in $2^{O(|\varphi|)}$, $\Sigma = 2^{Prop}$, and $\mathscr{L}_\omega(A_\varphi)$ is exactly $\models \varphi$.

PROOF OF THEOREM 2.1[6]

For simplicity, presume φ is in negation normal form ($\neg$ appears only preceding propositions) and the temporal operators $\square$ and $\diamond$ have been eliminated via the equivalences described in Chapter 2.2.1.

Recall that the closure of LTL formula φ, $cl(\varphi)$ is the set of all of the subformulas of φ and their negations (with redundancies such as φ and $\neg\neg\varphi$ consolidated). Specifically,

- $\varphi \in cl(\varphi)$.

- $\neg\psi \in cl(\varphi) \rightarrow \psi \in cl(\varphi)$.

- $\psi \in cl(\varphi) \rightarrow \neg\psi \in cl(\varphi)$.

- $\xi \wedge \psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.

- $\xi \vee \psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.

- $X\,\psi \in cl(\varphi) \rightarrow \psi \in cl(\varphi)$.

- $\xi\,\mathcal{U}\,\psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.

- $\xi\,\mathcal{R}\,\psi \in cl(\varphi) \rightarrow \xi, \psi \in cl(\varphi)$.

---

[6]The proof here was inspired by the direct proof techniques employed in [43] (which proved a variant of Theorem 2.1 that reasons more directly over graphs instead of Büchi automata), [130], and [33]. Other proof strategies include the use of alternating automata [88, 131]. Alternatively, one can consider the Büchi automaton A_φ to be a combination of two automata, a local automaton that reasons about consecutive sequences of states and an eventuality automaton that reasons about eventuality formulas ($\mathcal{U}-$ or $\mathcal{F}$-subformulas) [39, 44, 132, 133]. (Note that [44, 132, 133] proved a variant of Theorem 2.1 for types of Extended Temporal Logic (ETL) that subsume LTL.)

Recall that the *expansion laws* state that $(\xi\ \mathcal{U}\ \psi) = \psi \vee [\xi \wedge \mathcal{X}(\xi\ \mathcal{U}\ \psi)]$ and $(\xi\ \mathcal{R}\ \psi) = \psi \wedge [\xi \vee \mathcal{X}(\xi\ \mathcal{R}\ \psi)]$; we can use these laws to construct an elementary set of formulas. An *elementary set* of formulas with respect to the closure of φ is a maximal, consistent subset of $cl(\varphi)$. A *cover* C of a set of formulas Ψ is a set of sets $C = C_0, C_1, \ldots$ such that $\bigwedge_{\psi \in \Psi} \psi \leftrightarrow \bigvee_{C_i \in C} \bigwedge_{\gamma \in C_i} \gamma$. In other words, any computation satisfying the conjunction of the formulas in the set Ψ also satisfies the conjunction of the formulas in at least one of the cover sets $C_i \in C$. An *elementary cover* is a cover comprised exclusively of elementary (maximal, consistent) sets of formulas.

We define each elementary cover set $C_i \in C$ for the cover C of formula φ to have the following properties:

1. $C_i \subseteq cl(\varphi)$

2. C_i is logically consistent (i.e. does not contain any logical contradictions). Specifically, for all subformulas $\xi, \psi \in cl(\varphi)$,

   - $\psi \in C_i \leftrightarrow \neg\psi \notin C_i$

   - $\xi \wedge \psi \in C_i \leftrightarrow \xi, \psi \in C_i$.

   - $\xi \vee \psi \in C_i \leftrightarrow (\xi \in C_i)$ or $(\psi \in C_i)$.

3. C_i is temporally consistent (i.e. logically consistent with respect to temporal operators). We use $\Rightarrow$ to denote subformulas that are *syntactically implied* by a set C_i.

   - $(\xi\ \mathcal{U}\ \psi \in cl(\varphi)) \rightarrow [(\psi \in C_i) \Rightarrow (\xi\ \mathcal{U}\ \psi \in C_i)]$.

   - $[(\xi\ \mathcal{U}\ \psi \in C_i) \wedge (\psi \notin C_i)] \rightarrow (\xi \in C_i)$.

   - $(\xi\ \mathcal{R}\ \psi \in cl(\varphi)) \rightarrow [(\psi \in C_i) \Rightarrow (\xi\ \mathcal{R}\ \psi \in C_i)]$.

4. C_i is maximal. That is, for every subformula $\psi \in cl(\varphi)$, either $\psi \in C_i$ or $\neg\psi \in C_i$.

We can obtain an elementary cover of φ by applying the expansion laws to φ until φ is a propositional formula in terms of only constants (*true* or *false*), propositions, or X-rooted subformulas. Then φ has no $\mathcal{U}$- or $\mathcal{R}$-formulas occurring at the top level. We convert this expanded formula into disjunctive normal form (DNF) to obtain the cover. Each disjunct is an elementary set and the set of disjuncts is the elementary cover of φ. We round out each set C_i with the formulas in $cl(\varphi)$ that are syntactically implied by C_i according to the rules listed above. We define a state in our automaton A_φ for each cover set C_i in the cover of φ. We label each such state with the subformulas in the elementary set to which it corresponds. (Note that the states in the cover of φ will be labeled by φ.) We will use the X-subformulas of each state to define the transition relation.

Define q_0 as the state or set of states in the cover of φ (i.e. those labeled by φ). If there is more than one such state and we do not want to define q_0 to be a set of initial states, we can create a singular start state with outgoing λ-transitions[7] to all such states. That is, for φ-labeled states $s_0, s_1, \ldots$, we define $\delta(q_0, \lambda) = s_0; \delta(q_0, \lambda) = s_1; \ldots$.

Initially, we set $Q = q_0$, which is the cover of φ. For each state $q \in Q$, we apply the expansion laws and compute the successors of q as a cover of $\{\psi : X\psi \in q\}$. For each computed successor state q' of q, we add a transition in δ, creating a new state $q' \in Q$ if necessary. That is, $\forall q', \sigma : \delta(q, \sigma) = q', X\psi \in q \rightarrow \psi \in q'$. We iterate in this fashion until all X-subformulas of all of the states in Q have been covered. The result is a *closed set of elementary covers*, such that there is an elementary cover in the set for each X-subformula of each disjunct of each cover in the set.[8] Constructed in this way, Q is at most the set of all elementary sets of formulas in the closure of φ ($2^{cl(\varphi)}$) and is thus bounded by $2^{O(|\varphi|)}$.

---

[7]We refer to λ as the empty string. The presence of a λ-transition in a nondeterministic automaton indicates that the automaton may traverse that transition at any time, without progressing any further along the computation path.

[8]Note that any LTL formula has infinitely many elementary covers and which one is chosen for this construction in practice can significantly affect the performance of the model-checking problem. Thus, many researchers study optimizing this construction [134, 135, 136, 137, 138, 139, 140, 141].

Finally, we define the acceptance conditions, ensuring that each $\mathcal{U}$-subformula is eventually fulfilled and not simply promised forever. We do this by creating for every subformula of φ of the form $\xi\,\mathcal{U}\,\psi$ a set $F_{\xi\,\mathcal{U}\,\psi} \in F$ containing all states labeled by ψ, which fulfill the $\mathcal{U}$-subformula, and all states not labeled by $(\xi\,\mathcal{U}\,\psi)$, which do not promise it. This is the basic construction by Daniele, Giunchiglia, and Vardi [135].

We prove each direction separately based upon these definitions.

If: $\pi \in \mathcal{L}_\omega(A_\varphi) \rightarrow \pi \models (\varphi)$

Presume $\pi \in \mathcal{L}_\omega(A_\varphi)$. Then there is an accepting run in A_φ that we will label $\rho = q_0, q_1, \ldots$ which, by definition of Q, corresponds to the sets $C_0, C_1, \ldots$. By the definition of q_0 as the cover of φ, we know that φ holds in this state. Stated another way, the atomic propositions that are true in q_0 are true in the first time step of a satisfying computation of φ; $\pi_0 \in C_0$. It remains to show that the transition relation and acceptance conditions imply that $\pi \models (\varphi)$, or, in other words, $(C_0 \cap Prop)(C_1 \cap Prop)(C_2 \cap Prop) \ldots \models \varphi$. We show by structural induction on the structure of $\psi \in cl(\varphi)$ that $\psi \in C_0 \leftrightarrow \pi \models (\psi)$.

Base case: $|\psi| = 1$. Then $\psi \in Prop$. The claim follows from the definition of the labeling of states in Q, i.e. that in our construction, $C_i = \{\psi \in cl(\varphi) | (C_i \cap Prop)(C_{i+1} \cap Prop)(C_{i+2} \cap Prop) \ldots \models \psi\}$.

Induction step: Presume the claim holds for subformulas $\xi, \eta \in cl(\varphi)$. Then it holds for:

1. $\psi = \neg\xi$. The claim follows from the definition of the labeling of states in Q.

2. $\psi = \mathcal{X}\xi$. From the definition of δ, $\mathcal{X}\xi \in C_i \rightarrow \forall q_{i+1}, \sigma : \delta(q_i, \sigma) = q_{i+1}, \xi \in C_{i+1}$. Therefore, $\psi \in C_i \leftrightarrow \pi \models (\psi)$.

3. $\psi = \xi \wedge \eta$. The claim follows from the definition of the labeling of states in Q.

4. $\psi = \xi \vee \eta$. The claim follows from 1, 3, and DeMorgan's law.

5. $\psi = \xi\,\mathcal{U}\,\eta$. If $\pi \models \psi$ then there is some $i \geq 0$ such that $\pi_i, \pi_{i+1}, \ldots \models \eta$ and

$\forall j : 0 \leq j < i, \pi_j, \pi_{j+1}, \ldots \models \xi$. From our induction hypothesis, we have that $\eta \in C_i$ and $\xi \in C_j$. By induction we have that $\xi \, \mathcal{U} \, \eta \in C_i, C_{i-1}, \ldots, C_0$. From our construction, if $(\xi \, \mathcal{U} \, \eta) \in C_0$ then either $\forall i \geq 0 : \xi, \mathcal{X}(\xi \, \mathcal{U} \, \eta) \in C_i$ and $\eta \notin C_i$ or $\exists i \geq 0 : \eta \in C_i$ and $\forall j : 0 \leq j \leq i, \xi, \mathcal{X}(\xi \, \mathcal{U} \, \eta) \in C_i$. Since we know that ρ, the run of π in A_φ, is accepting, by the definition of F, only the latter case is possible. Therefore, $\psi \in C_0 \leftrightarrow \pi \models (\psi)$.

6. $\psi = \xi \mathcal{R} \eta$. The claim follows from 1, 5, and the definition of $\mathcal{R}$.

Only if: $\pi \models (\varphi) \rightarrow \pi \in \mathscr{L}_\omega(A_\varphi)$

Since q_0 is defined by the cover of φ, there must be a state $q \in q_0$ such that $\pi_0 \models q$. In general, we want to show that $\forall i : i \geq 0, \pi_i \models q_i$, so we show how to choose π_{i+1}. We know there is a set $C_{i+1} \in \delta(C_i, \pi_i)$ such that $\pi_i, \pi_{i+1}, \ldots \models C_i, C_{i+1}, \ldots$ since for all i:

- The atomic propositions that are true in π_0 are true in the state q_i of our run: $\pi_i \in C_i$.

- For every formula of the form $\mathcal{X}\psi$ in C_i, we know that $\pi_i, \pi_{i+1}, \ldots \models \mathcal{X}\psi$, which means $\pi_{i+1}, \pi_{i+2}, \ldots \models \psi$, so $\psi \in C_{i+1}$.

- For every formula of the form $\xi \, \mathcal{U} \, \psi$ in C_i, we know that $\pi_i, \pi_{i+1}, \ldots \models \xi \, \mathcal{U} \, \psi$, which means either $\pi_i, \pi_{i+1}, \ldots \models \psi$, so $\psi \in C_i$, or $\pi_i, \pi_{i+1}, \ldots \models \xi \wedge \mathcal{X}(\xi \, \mathcal{U} \, \psi)$, so $\xi \in C_i$ and $(\xi \, \mathcal{U} \, \psi) \in C_{i+1}$.

Furthermore, we know from the definition of F that for all subformulas of the form $(\xi \, \mathcal{U} \, \psi) \in cl(\varphi)$, there is a set $F_{\xi \, \mathcal{U} \, \psi} \in F$ containing all $C_i : (\psi \in C_i) \vee ((\xi \, \mathcal{U} \, \psi) \notin C_i)$. We can choose an accepting run as follows. Let U be the set of $\mathcal{U}$-subformulas not fulfilled in the current state, C_i. Then $\forall \xi, \psi : ((\xi \, \mathcal{U} \, \psi) \in C_i) \wedge \psi \notin C_i, (\mathcal{X}(\xi \, \mathcal{U} \, \psi) \in C_i)$. For each such formula in U in succession, we choose the shortest path from the current state to a state that contains ψ, thus fulfilling the claim. We know there must exist such a path for each $\mathcal{U}$-subformula by the construction of δ such that $\mathcal{X}\psi \in C_i \rightarrow \psi \in C_{i+1}$, and by the definition of F. Note that

if some state $C_i \notin F_{\xi\,\mathcal{U}\,\psi}$, then $((\xi\,\mathcal{U}\,\psi) \in C_i)$ and $(\psi \notin C_i)$ and, also, $\pi_i, \pi_{i+1}, \ldots \models (\xi\,\mathcal{U}\,\psi)$ but $\pi_i, \pi_{i+1}, \ldots \not\models \psi$. This creates a contradiction since we know that $\pi \models (\varphi)$, by defini-tion of Büchi acceptance, necessitates that π passes through a state in each set $F_{\xi\,\mathcal{U}\,\psi} \in F$ infinitely often. ∎

This theorem reduces LTL satisfiability checking to automata-theoretic nonemptiness checking, as φ is satisfiable iff $models(\varphi) \neq \emptyset$ iff $\mathscr{L}_\omega(A_\varphi) \neq \emptyset$.[9]

2.5   Safety versus Liveness

There are two fundamental properties we can prove about programs: *safety* and *liveness*. These categories were originally introduced by Leslie Lamport [142]. They are also re-ferred to as *invariance* and *eventuality*, respectively [86]. Distinguishing between the two types of properties is helpful because different techniques can be used to prove each type [143, 144]. For example, taking advantage of the structure of safety properties can be used to optimize assume-guarantee reasoning [144]. Showing a safety property holds involves an invariance argument while liveness properties require demonstration of well-foundedness. Here, we adapt the formal definitions of safety and liveness from Alpern and Schneider [145]. Kindler [146] provides a complete survey of historical (and sometimes conflicting) definitions.

Intuition 1

A **Safety Property** *expresses the sentiment that "something bad never happens."*

In general, safety properties take the form □*good*, where *good* is a temporal logic for-mula describing a good system behavior. In other words, the system always stays in some allowed region of the state space. Recall that LTL properties are interpreted over computa-tions, which are infinite words over the alphabet $\Sigma = 2^{Prop}$ where *Prop* is the set of atomic

---

[9]For another version of this proof with several illustrative examples, see [33].

propositions in the formula. Intuitively, "something bad" only needs to happen once in a computation for the property to be violated; what happens after that in the infinite computation does not affect the outcome of the model-checking step. Therefore, if a safety property φ is violated, there must be some finite prefix $\alpha = \pi_0, \pi_1, \ldots \pi_i$ of the computation π in which this violation occurs. Then, for any infinite computation β over the alphabet Σ, the concatenation of computations $\alpha \cdot \beta$ cannot satisfy φ.

Safety properties were first formally defined in [147]. For property φ, infinite computation $\pi = \pi_0, \pi_1, \ldots$, and alphabet $\Sigma = 2^{Prop}$, we define safety as follows:

$$(\forall \pi, \pi \in \Sigma^\omega : \pi \models \varphi \leftrightarrow (\forall i, 0 \leq i : (\exists \beta, \beta \in \Sigma^\omega : \pi_{0\ldots i} \cdot \beta \models \varphi))),$$

where $\cdot$ is the concatenation operator and Σ^ω denotes the set of infinite words (or ω-words) over Σ.

Restated, φ is a safety property satisfied by computation π if and only if for all lengths of finite prefixes of π, that finite prefix concatenated with some infinite computation models φ. This is the contrapositive of the statement that if $\pi \not\models \varphi$ then there is some finite prefix of π in which something bad happens that cannot be extended into an infinite computation that will satisfy φ.

We can also define a safety property in terms of language closure [145]. In Chapter 2.4 we translate the LTL formula into a Büchi automaton. The closure of a reduced Büchi automaton is obtained by making all states accepting states. (A Büchi automaton is reduced if there is a path to an accepting state from every state in the automaton; i.e. there are no dead end "trap" states.) A closure of an automaton representing a safety property must accept all prefixes of the language defined by that property.

Lemma 2.2

A reduced Büchi automaton A_φ specifies a safety property φ if and only if $\mathscr{L}(A_\varphi) =$

$\mathscr{L}(cl(A_\varphi)).$[10]

Examples of safety properties include invariance, partial correctness, mutual exclusion, ordering (i.e. first-come-first-serve), deadlock freedom, and reachability. Safety properties are also called invariance properties because they describe predicates that do not change under a set of transformations. Proving partial correctness of a sequential program provides assurance that the program never halts with a wrong answer. Mutual exclusion prohibits the simultaneous use of a common resource by multiple processes (i.e. two processes cannot write to the same file at the same time). Ordering of events, such as first-come-first-served, forbids service out of the established order. Freedom from deadlock ensures that the system will never reach a standstill from which no progress can be made, such as when all processes are simultaneously locked out of, and waiting on, a common resource. In a sense, reachability is the dual of safety as it is an assurance that a certain program state can be reached (i.e. $\neg(\Box\neg good)$).

Intuition 2

*A **Liveness Property** expresses the sentiment that "something good must eventually happen."*

Typically, liveness properties express $\Diamond good$, where *good* once again is a temporal logic formula describing a good system behavior. Whereas safety properties reason about reaching specific states, liveness properties reason about control flow between states. Liveness is loosely defined as a program's ability to execute in a timely manner.

Let Σ^{finite} be the set of all finite computations over Σ. For property φ, we define liveness as follows:

$$(\forall\alpha, \alpha \in \Sigma^{finite} : (\exists\beta, \beta \in \Sigma^\omega : \alpha \cdot \beta \models \varphi)),$$

---

[10]Note that Sistla showed that the problem of determining whether an LTL formula, or the nondeterministic Büchi automaton representing it, express a safety property is PSPACE-complete [148].

where · is the concatenation operator.

The formal definiton of liveness is basically defined oppositely from safety. In short, there is no finite prefix over Σ that cannot be extended into an infinite computation that satisfies φ. Intuitively, this corresponds to the notion that the "something good" that satisfies the property can still happen after any finite execution.

As with safety, we can also frame the definition of liveness in terms of language closure of reduced Büchi automata. If A_φ is a reduced Büchi automaton then a computation is not in $cl(A_\varphi)$ if and only if it attempts to traverse a transition that is not defined in the transition relation δ for A_φ. Furthermore, since every finite prefix of liveness property φ can be extended into an infinite computation that satisfies φ, then the closure of A_φ actually accepts all infinite words over the alphabet Σ.

Lemma 2.3

A reduced Büchi automaton A_φ specifies a liveness property φ if and only if $\mathscr{L}(cl(A_\varphi)) = \Sigma^\omega$.

Liveness properties are also called eventualities since they promise that some event will happen, eventually. In other words, the system will eventually progress through "useful" states, even if some stuttering is allowed. Termination is a liveness property; it asserts the program will eventually halt instead of hanging forever. Other examples of liveness properties include total correctness, accessibility/absence of starvation, fairness (also called justice), compassion (also called strong fairness), and livelock freedom. Total correctness extends partial correctness with termination; it states that not only must the program not terminate in a bad state, the program must terminate in a good state. Starvation occurs when a process is unable to make progress due to insufficient access to shared resources; it is assuaged by assuring regular accessibility to necessary resources. Fairness guarantees each process the chance to make progress while compassion extends fairness to include the

existence of sets of cyclically recurring system states. A fair scheduler executes a process infinitely often while a compassionate scheduler ensures that a process that is enabled infinitely often is executed infinitely often. Freedom from livelock means that the system will never reach a *live* standstill, where the processes in question cannot make progress yet are not blocked but thrashing indefinitely. An intuitive example of livelock occurs when two people try to pass each other in the hallway yet repeatedly step to the same side of the hallway in an effort to let the other person pass. Both people are still walking but they are now oscillating side-to-side instead of making progress forward. Common examples of liveness properties include: "every request will eventually be granted," "a message will eventually reach its destination," and "a process will eventually enter its critical section."

Model checking of safety properties is simpler than model checking of liveness properties. Due to the temporal structure of liveness properties, they must be witnessed by an infinite counterexample in the shape of a lasso. That is, a counterexample with a finite prefix leading to a bad cycle that represents an unending run of the system in which the liveness property is never fulfilled. Safety properties, on the other hand, are always violated within the finite prefix, eliminating the need to compute the remainder of the counterexample corresponding to the loop at the end of the lasso. Intuitively, a safety property is violated the first instance the system enters some prohibited state. Though we provide the full model-checking algorithm for all possible LTL specifications in Chapter 2.10, a simpler check using symbolic reachability analysis can be used instead to check only safety properties [144].

With only one exception, safety and liveness properties are disjoint. The only property that is both a safety and a liveness property is the property *true*, which describes all possible behaviors, or the words in the set $(2^{Prop})^\omega$.[11] The proof of this statement follows from

---

[11]Note ω was originally defined by Cantor to be the lowest transfinite ordinal number so $(2^{Prop})^\omega$ designates the set of all infinite words over the alphabet $\Sigma = 2^{Prop}$.
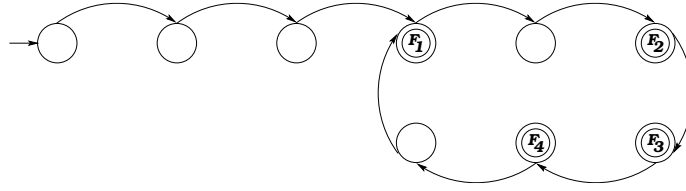
Figure 2.5 : A counterexample trace takes the form of an accepting lasso. Again, the start state is designated by an incoming, unlabeled arrow not originating at any vertex. Here, $F_1 \ldots F_4$ represent final states satisfying four acceptance conditions.

the definitions of safety and liveness above, specifically $\mathscr{L}(true) = \mathscr{L}(cl(true)) = \Sigma^{\omega}$. Remarkably, while every LTL formula is not strictly either a safety or a liveness property, Alpern and Schneider [149] proved that every LTL formula is a combination of a safety and a liveness property.

2.5.1 Specifying the Property as an Automaton

Instead of using a temporal logic to describe a behavioral property, we can use an automaton directly. Many model checkers, like Spin, NuSMV, and CadenceSMV, will accept specifications in either format and some, like Lucent's FormalCheck [150], deal exclusively with automata inputs. However, complementing a Büchi automaton is quite difficult (and likely involves the expensive step of determinization) whereas complementing an LTL formula can be done in constant time by simply preceding the formula with a negation symbol. Currently, the best time complexity for complementation of Büchi automata is $2^{O(n \log n)}$ [151] but this is still an open area of research [152, 153, 154]. Advantages of this method include the ability to directly and efficiently construct the automaton A_φ rather than relying on an LTL-to-Büchi translation while disadvantages include the unintuitiveness of representing a behavior specification as an automaton.

2.6 LTL → Symbolic GBA

Given an LTL specification φ, the model checker creates a generalized Büchi automaton $A_{\neg\varphi}$ that recognizes precisely the executions that do not satisfy φ and is destined for composition with the model under verification. The size of $A_{\neg\varphi}$ is $O(2^{|\varphi|})$ in the worse case, which provides motivation for utilizing a succinct, symbolic representation instead of explicitly constructing the automaton. Symbolic model checkers, such as NuSMV [20], CadenceSMV [14], SAL-SMC [76], and VIS [66], represent and analyze the system model and specifications symbolically. In the symbolic representation of automata, states are viewed as truth assignments to Boolean state variables and the transition relation is defined as a conjunction of Boolean constraints on pairs of current and next states [67]. All symbolic model checkers use the LTL-to-symbolic automaton translation described in [68], some with minor optimizations. Essentially, these tools support LTL model checking via the symbolic translation of LTL to a fair transition system that can be checked for nonemptiness. Recall that fairness constraints specify sets of states that must occur infinitely often in any path. They are necessary to ensure that the subformula ψ holds in some time step for specifications of the form $\xi \, \mathcal{U} \, \psi$ and $\Diamond\psi$. We enumerate the steps of the standard LTL-to-symbolic automaton algorithm [68], in detail, below. Bolded steps are those that produce output that appears in the symbolic automaton.

Input: An LTL formula f.

1. Negate f and **declare the set AP_f of atomic propositions of f**.

2. Build *el_list*, the list of elementary formulas in f:

   - $el(p) = \{p\}$ if $p \in AP$.

   - $el(\neg g) = el(g)$.

   - $el(g \vee h) = el(g) \cup el(h)$.

- $el(g \wedge h) = el(g) \cup el(h)$.

- $el(\mathcal{X}g) = \{\mathcal{X}g\} \cup el(g)$.

- $el(g \,\mathcal{U}\, h) = \{\mathcal{X}(g \,\mathcal{U}\, h)\} \cup el(g) \cup el(h)$.

- $el(g \,\mathcal{R}\, h) = \{\mathcal{X}(g \,\mathcal{R}\, h)\} \cup el(g) \cup el(h)$.

- $el(\Box g) = \{X(\Box g)\} \cup el(g)$.

- $el(\Diamond g) = \{X(\Diamond g)\} \cup el(g)$.

3. **Declare a new variable $EL_{\mathcal{X}_g}$ for each formula $\mathcal{X}g$ in the list _el_list_.**

4. Define the function $sat(g)$, which associates each elementary subformula g of f with each state that satisfies g:

   - $sat(g) = \{q | g \in q\}$ where $g \in el(f), q \in Q$.

   - $sat(\neg g) = \{q | q \notin sat(g)\}$.

   - $sat(g \vee h) = sat(g) \cup sat(h)$.

   - $sat(g \wedge h) = sat(g) \cap sat(h)$.

   - $sat(g \,\mathcal{U}\, h) = sat(h) \cup (sat(g) \cap sat(\mathcal{X}(g \,\mathcal{U}\, h)))$.

   - $sat(g \,\mathcal{R}\, h) = sat(h) \cap (sat(g) \cup sat(\mathcal{X}(g \,\mathcal{R}\, h)))$.

   - $sat(\Box g) = sat(g) \cap sat(\mathcal{X}(\Box g))$.

   - $sat(\Diamond g) = sat(g) \cup sat(\mathcal{X}(\Diamond g))$.

5. **Add fairness constraints to the SMV input model:**

$$\{sat(\neg(g \,\mathcal{U}\, h) \vee h) | g \,\mathcal{U}\, h \text{ occurs in } f\}.$$

$$\{sat(\neg(\Diamond g) \vee g) | \Diamond g \text{ occurs in } f\}.$$

Note that we do not add a fairness constraint for subformulas of the form $g \, \mathcal{R} \, h$. While $\mathcal{R}$ is the dual of $\mathcal{U}$, it is acceptable if h is true infinitely often and g is never true.

6. **Construct the characteristic function S_h of $sat(h)$. For each subformula h in $\neg f$:**

   $S_h = p$ if p is an atomic proposition.

   $S_h = EL_h$ if h is elementary formula $\mathcal{X}g$ in el_list.

   $S_h =! S_g$ if $h = \neg g$

   $S_h = S_{g1} | S_{g2}$ if $h = g_1 \vee g_2$

   $S_h = S_{g1} \& S_{g2}$ if $h = g_1 \wedge g_2$

   $S_h = S_{g2} | (S_{g1} \& S_{X_{(g1 \, \mathcal{U} \, g2)}})$ if $h = g_1 \, \mathcal{U} \, g_2$

   $S_h = S_{g2} \& (S_{g1} | S_{X_{(g1 \, \mathcal{R} \, g2)}})$ if $h = g_1 \, \mathcal{R} \, g_2$

   $S_h = S_g \& S_{X_{(\Box g)}}$ if $h = \Box g$

   $S_h = S_g | S_{X_{(\Diamond g)}}$ if $h = \Diamond g$

7. **For each subformula rooted at an $\mathcal{X}$, define a `TRANS` statement of the form:**

   - `TRANS ( S_X_g = next(S_g) )`

8. Print the SMV program. (Since we type code in ASCII text, the operators $\Box$ and $\Diamond$ are represented in SMV syntax by their alphabetic character equivalents, $\mathcal{G}$ and $\mathcal{F}$, respectively.)

```
MODULE main

VAR /*declare a boolean variable for each atomic proposition in f*/

a: boolean;
b: boolean;

/*and declare a new variable EL_X_g for each formula (X g) in el_list*/
EL_X_g: boolean;
```

```
DEFINE  /*for each S_h in the characteristic function, put a line here*/
  S_a := a;
  S_b := b;
  S_g1 := ...
  S_g2 := ...

TRANS   /*for each (X g) in el_list, generate a line here*/
  ( S_X_g1 = next(S_g1) ) &
  ( S_X_g2 = next(S_g2) ) &
         ...
  ( S_X_gn = next(S_gn) )

/*for each (g U h) and (F g) in the parse tree,
                    generate lines like this:*/
FAIRNESS   ! S_gUh | S_h
FAIRNESS   ! S_Fg | S_g

/*end with a SPEC statement*/
SPEC       !(S_f & EG true)
```

Proof of the correctness of this LTL-to-symbolic-automaton construction is given in [68].

2.7   Combining the System and Property Representations

We directly construct the product $A_{M,\,\neg\varphi}$ of the system model automaton M and the automaton representing the complemented specification $A_{\neg\varphi}$. This construction logically follows from the realization that the product of these two automata constitutes the intersection of the languages $\mathscr{L}(M)$ and $\mathscr{L}(A_{\neg\varphi})$ [155]. Recall from Chapter 2.2.2 that Büchi automata are automata that accept exactly the class of ω-regular languages. Furthermore, the class of ω-regular languages corresponds exactly to the languages that can be described by ω-regular expressions and is closed under the operations of union, intersection, and complementation [156]. Therefore, we can construct a Büchi automaton $A_{M,\,\neg\varphi}$ such that $\mathscr{L}(A_{M,\,\neg\varphi}) = \mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$ since $\mathscr{L}(A_{M,\,\neg\varphi})$ is an ω-regular language.

Intuitively, $A_{M,\,\neg\varphi}$ simulates running the two multiplicand automata in parallel, also called the *synchronous parallel composition* of M and $A_{\neg\varphi}$. Executing these two automata

simultaneously allows us to use $A_{\neg\varphi}$ to continuously monitor the behavior of M and judge whether M satisfies the desired property φ at all times.[12]

The states of $A_{M,\,\neg\varphi}$ are pairs of states, one each from M and $A_{\neg\varphi}$, plus a track label $t \in \{0, 1\}$. (For simplicity, we call a state in M q_M and a state in $A_{\neg\varphi}$ q_φ.) A run of $A_{M,\,\neg\varphi}$ starts in the pair of the two start states in track 0: $(q_{0_M}, q_{0_\varphi}, 0)$. Upon reading a character $\sigma \in \Sigma$, $A_{M,\,\neg\varphi}$ transitions from the current state, $q = (q_M, q_\varphi, t)$, to a next state, $q' = (q'_M, q'_\varphi, t)$, wherever M transitions from the M-component of q, q_M, to the M-component of q', q'_M, upon reading σ and $A_{\neg\varphi}$ also transitions from its associated component, q_φ, to q'_φ upon reading σ. The value of t remains the same unless either $t = 0$ and $q_M \in F_M$ or $t = 1$ and $q_\varphi \in F_\varphi$, in which case the t-bit flips. In essence, the two tracks ensure $A_{M,\,\neg\varphi}$ starts in track 0, passes through a final state in M, switches to track 1, passes through a final state in $A_{\neg\varphi}$, switches back to track 0, and repeats the pattern. We assign the set of final states in $A_{M,\,\neg\varphi}$ to essentially match those of M since the path of any run only returns to track 0 if it has previously passed through an accepting state of M in track 0 and an accepting state of $A_{\neg\varphi}$ in track 1. Since the accepting condition for a Büchi automaton necessitates visiting the set of final states infinitely often, visiting infinitely often a state whose M-component is in F_M while in track 0 suffices. This construction is necessarily more complex than the straight *Cartesian product* [155] of the two automata, $M \times A_{\neg\varphi}$, since we must allow for runs that pass infinitely often through final states in both M and $A_{\neg\varphi}$, but not necessarily at the same time. (For an illustrative example, see [131].) Note that since q_{0_φ} is defined as the state(s) labeled by $\neg\varphi$, the initial state(s) our product automaton $A_{M,\,\neg\varphi}$ must also be labeled by $\neg\varphi$.

We presume that the alphabets of both automata are the same, a reasonable assumption considering that the formula $\neg\varphi$ describes a behavior of the system M. However, our con-

---

[12]Systems can also be composed asynchronously, where δ is defined such that it is possible for either M or φ to progress while the other does not transition and the values of any system variable specific to the non-transitioning system are preserved. However, this type of composition does not further the model-checking algorithm presented here.

struction is unchanged by defining the alphabet Σ of $A_{M, \neg\varphi}$ to be the union of the alphabets of M and $A_{\neg\varphi}$ if those alphabets differ.

Formally, we define $A_{M, \neg\varphi}$ as follows.

Definition 6

Let $M = (Q_M, \Sigma, \delta_M, q_{0M}, F_M)$ and $A_{\neg\varphi} = (Q_\varphi, \Sigma, \delta_\varphi, q_{0\varphi}, F_\varphi)$. The intersection automaton $A_{M, \neg\varphi}$ with the property that $\mathscr{L}(A_{M, \neg\varphi}) = \mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$ is the automaton defined by the quintuple $A_{M, \neg\varphi} = (\hat{Q}, \Sigma, \hat{\delta}, (q_{0M}, q_{0\varphi}, 0), \hat{F})$ where:

- *State set $\hat{Q} = Q_M \times Q_\varphi \times \{0, 1\}$ consists of triples $(q_{iM}, q_{j\varphi}, t)$.*

- *Σ is the shared alphabet.*

- *Transition relation $\hat{\delta}$ is defined such that $A_{M, \neg\varphi}$ is in state $(q_{iM}, q_{j\varphi}, t)$ whenever M is in state q_{iM} and $A_{\neg\varphi}$ is in state $q_{j\varphi}$. For any $\sigma \in \Sigma$,*

$$\hat{\delta}((q_{iM}, q_{j\varphi}, t), \sigma) = (q_{kM}, q_{l\varphi}, t),$$

whenever

$$\delta_M(q_{iM}, \sigma) = q_{kM}$$

and

$$\delta_\varphi(q_{j\varphi}, \sigma) = q_{l\varphi}$$

unless

$$t = 0 \text{ and } q_{iM} \in F_M,$$

or

$$t = 1 \text{ and } q_{j\varphi} \in F_\varphi$$

in which case

$$\hat{\delta}((q_{iM}, q_{j\varphi}, t), \sigma) = (q_{kM}, q_{l\varphi}, \bar{t}).$$

- *The initial state is $(q_{0M}, q_{0\varphi}, 0)$.*

- *The set of final states $\hat{F}$ is the set of all states $(q_{iM}, q_{j\varphi}, 0)$ such that $q_{iM} \in F_M$ and $q_{j\varphi} \in Q_\varphi$.*

Theorem 2.2

Büchi automata are closed under complementation.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a Büchi automaton that accepts the language L. The construction of the Büchi automaton $\overline{A}$ that accepts the complement language $\overline{L}$ is rather involved. It does not suffice to simply complement the accepting set F (i.e. let $A' = (Q, \Sigma, \delta, q_0, Q - F)$) because a run not passing through any state in F infinitely often is different from a run passing through some state in $Q - F$ infinitely often. For example, an accepting run of A on the word w might pass through both states in F and in $Q - F$ infinitely often, so both A and A' would accept w. Thus, $\mathcal{L}(A') \neq \Sigma^\omega - \mathcal{L}(A)$. The complex construction proving Theorem 2.2 is given in [129], which yields a doubly exponential construction for $\overline{A}$ and in [107], which reduces the construction to singly exponential with a quadratic exponent. Specifically, for Büchi automaton A, over the alphabet Σ, there is a Büchi automaton $\overline{A}$ with $2^{O(|Q| \log |Q|)}$ states such that $\mathcal{L}(\overline{A}) = \Sigma^\omega - \mathcal{L}(A)$ [151]. Michel proved that this is an optimal bound [157].

As an important aside, since Büchi automata are not closed under determinization and nondeterministic Büchi automata are more expressive than deterministic Büchi automata, the complement automaton $\overline{A}$ produced by these algorithms for some deterministic Büchi automaton A may be a nondeterministic Büchi automaton that cannot be determinized. Efficient methods for constructing and reasoning about complemented Büchi automata remain an area of interest. For example, we can check A for nonuniversality (i.e. $\mathcal{L}(A) \neq \Sigma^\omega$) in polynomial space by constructing $\overline{A}$ on the fly and checking for nonemptiness [131].

Theorem 2.3

$\mathcal{L}(A_{M, \neg\varphi})$ *is an ω-regular language and $A_{M, \neg\varphi}$ is a Büchi automaton.*

PROOF OF THEOREM 2.3

Recall that $L_M = \mathcal{L}(M)$ is an ω-regular language since it is the language accepted by

a Büchi automaton and that $L_\varphi = \mathscr{L}(A_{\neg\varphi})$ is also an ω-regular language since $\neg\varphi$ is an LTL formula, which describes a ($*$-free) ω-regular expression. We know that $\mathscr{L}(A_{M,\,\neg\varphi}) = L_M \cap L_\varphi$ is an ω-regular language, and, therefore, that $A_{M,\,\neg\varphi}$ is a Büchi automaton because ω-regular languages are closed under intersection. First, we offer proof of closure under union; closure under intersection follows from closure under union and complementation.

By definition of ω-regular languages, there exist ω-regular expressions r_M and r_φ that describe the languages L_M and L_φ, respectively. That is, $L_M = \mathscr{L}(r_M)$ and $L_\varphi = \mathscr{L}(r_\varphi)$. By definition of ω-regular expressions, $r_M + r_\varphi$ is an ω-regular expression denoting the language $L_M \cup L_\varphi$, which demonstrates closure under union.

Closure under intersection now follows from DeMorgan's Law:

$$L_M \cap L_\varphi = \overline{\overline{L_M} \cup \overline{L_\varphi}}$$

By closure under complementation, $\overline{L_M}$ and $\overline{L_\varphi}$ are both ω-regular languages. By closure under union, $\overline{L_M} \cup \overline{L_\varphi}$ is an ω-regular language. Again by closure under complementation,

$$\overline{\overline{L_M} \cup \overline{L_\varphi}} = L_M \cap L_\varphi$$

is an ω-regular language. By definition of ω-regular languages, it follows that $A_{M,\,\neg\varphi}$ is a Büchi automaton. ∎

Theorem 2.4

Let $L_M = \mathscr{L}(M)$ and $L_\varphi = \mathscr{L}(A_{\neg\varphi})$. Given an infinite word $w = w_0, w_1, w_2, \ldots \in \Sigma$, $w \in L_M \cap L_\varphi$ if and only if it is accepted by $A_{M,\,\neg\varphi}$.

Proof of Theorem 2.4

If-direction: $w \in L_M \cap L_\varphi \rightarrow w$ is accepted by $A_{M,\,\neg\varphi}$.

An infinite word w is in the language $L_M \cap L_\varphi$ if $w \in L_M$ and $w \in L_\varphi$. By definition, $w \in L_M$ iff w represents an accepting execution of the Büchi automaton M. Symmetrically,

$w \in L_\varphi$ iff w is accepted by the automaton $A_{\neg\varphi}$ corresponding to the LTL formula $\neg\varphi$. Then we know that $\delta_M^\omega(q_{0M}, w)$ transitions through some final state $q_{iM} \in F_M$ infinitely often and that $\delta_\varphi^\omega(q_{0\varphi}, w)$ similarly transitions through some accepting state $q_{f\varphi} \in F_\varphi$ infinitely often. By definition of $\hat{\delta}$, $A_{M,\neg\varphi}$ is in state $(q_{iM}, q_{j\varphi}, t)$ for some $t \in \{0, 1\}$ exclusively when M is in state q_{iM} and $A_{\neg\varphi}$ is in state $q_{j\varphi}$ upon reading the same word. That is, for any w, t,

$$\hat{\delta}^\omega((q_{iM}, q_{j\varphi}, t), w) = (q_{kM}, q_{l\varphi}, t),$$

whenever

$$\delta_M^\omega(q_{iM}, w) = q_{kM}$$

and

$$\delta_\varphi^\omega(q_{j\varphi}, w) = q_{l\varphi}.$$

Since whenever $q_{iM} \in F_M$, $(q_{iM}, q_{j\varphi}, 0) \in \hat{F}$, and a state $(q_{iM}, q_{j\varphi}, 0)$ can only be reached when $\delta_\varphi(q_{0\varphi}, w)$ loops through some final state $q_{f\varphi} \in F_\varphi$, w is accepted by $A_{M,\neg\varphi}$.

Only-if direction: w is accepted by $A_{M,\neg\varphi} \rightarrow w \in L_M \cap L_\varphi$.

By definition, an infinite word w is accepted by $A_{M,\neg\varphi}$ if the run over w in $A_{M,\neg\varphi}$ visits at least one state in $\hat{F}$ infinitely often. That is, $\hat{\delta}^\omega((q_{0M}, q_{0\varphi}, 0), w)$ transitions infinitely often through some state $(q_{iM}, q_{j\varphi}, 0) \in \hat{F}$. By construction, for any w, $\hat{\delta}^\omega((q_{0M}, q_{0\varphi}, 0), w) = (\delta_M^\omega(q_{0M}, w), \delta_\varphi^\omega(q_{0\varphi}, w), 0)$. By definition of $\hat{F}$, $(q_{iM}, q_{j\varphi}, 0) \in \hat{F}$ iff $q_{iM} \in F_M$ and a run of w in $A_{\neg\varphi}$ loops through $q_{f\varphi} \in F_\varphi$. Therefore, $\delta_M^\omega(q_{0M}, w)$ transitions through $q_{iM} \in F_M$ infinitely often and $w \in L_M$. Symmetrically, $\delta_\varphi^\omega(q_{0\varphi}, w)$ transitions through $q_{f\varphi} \in F_\varphi$ infinitely often and $w \in L_\varphi$. Ergo, $w \in L_M \cap L_\varphi$. ∎

The product automaton $A_{M,\neg\varphi}$ has size $O(|M| \times |A_{\neg\varphi}|)$ in the worst case. (Though the product can be much smaller; in the best case, it has size 0 [59].) Due to this size blow-up, representing the automaton as succinctly as possible becomes vital to palliate the state explosion problem and ensure $A_{M,\neg\varphi}$ can be stored in computer memory. It is easy to see that even small differences in the sizes of both M and $A_{\neg\varphi}$ can have a significant impact on our ability to store, and to reason about, $A_{M,\neg\varphi}$.

2.8   Checking for Counterexamples: Explicitly

Explicit model checkers, such as Spin, search the combined automaton $A_{M, \neg\varphi}$ via a nested depth-first search algorithm [60]. The intuition behind this procedure is that the model checker must first search for a path from a start state to an accepting state and then search for a cycle from such an accepting state back to itself. Thus, the first search pauses when it reaches an accepting state and starts the second search to try and find a cycle through this state, resuming if the second search fails to find such a cycle.

The specific optimized nested depth-first search algorithm used in Spin and displayed in Figure 2.6 is a variation on the standard algorithm [60], which implements optimizations that both improve the performance of the search and ensure compatibility with partial order reduction [2]. Correctness in combination with partial order reduction is accomplished by guaranteeing that the second depth-first search always explores the same states that are found in the first depth-first search. The algorithm in Figure 2.6 provides this guarantee by terminating the second depth-first search whenever it reaches a target state that is on the `Stack` from the first depth-first search, because reaching such a target state from the state seeding the second depth-first search establishes the existence of a cycle. This cycle passes from the seed state to the target state (on the path explored via the second depth-first search), through the states on the `Stack` after the target state (explored via the first depth-first search), and back to the seed state.

In any given state, Spin always executes the actions from the equivalent state of the property automaton $A_{\neg\varphi}$ (represented as a Promela `never` claim) before those from the system M. Spin effectively combines the specification and system model on the fly in this way, minimizing the size of the `Statespace` that is constructed, and analyzing the possible combined `never` claim and model transitions using the optimized nested depth first search in Figure 2.6.

Optimized Nested Depth First Search (Compatible with Reduction) Used in Spin:

```
proc dfs(s)
   if error(s) then report error fi
   add {s,0} to Statespace
   add s to Stack
   for each (selected) successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
   od
   if accepting(s) then ndfs(s) fi
   delete s from Stack
end
proc ndfs(s) /* the nested search */
   add {s,1} to Statespace
   for each (selected) successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t in Stack then report cycle fi
   od
end
```

Figure 2.6 : Pseudocode representation of the NDFS algorithm from [2].

For model checking scalability, we must choose a representation of $A_{M, \neg\varphi}$ that minimizes the storage memory requirement while maximizing the efficiency of the nonemptiness check. An alternative to the above method of explicit representation and search by single states and transitions is to optimize this task by representing $A_{M, \neg\varphi}$ using BDDs.

2.9 Representing the Combined System and Property using BDDs

The basis for symbolic model checking is the realization that both the system model and the specification property can be represented symbolically, using Boolean equations, rather than explicitly, using automata. Furthermore, they can be manipulated efficiently using operations over Binary Decision Diagrams (BDDs). Symbolic representation and manipulation of the intermediate products of model checking leads to much more succinct structures needing to be stored in computer memory, thereby directly combating the state explosion

problem. Certainly there are pathological constructions of systems that do not benefit substantially from this space reduction, but symbolic model checking increases the scalability of model checking for a great many common systems, especially those with regular structure such as hardware circuits [81, 71].

In this section, we will demonstrate how the automaton $A_{M, \neg\varphi}$ can easily grow to a size that cannot be practically represented in memory and present the most popular alternative representation, which uses BDDs to mitigate this problem. The technique of *symbolic model checking* was introduced by McMillan [13] in 1992 specifically as a response to the state explosion problem. The power of symbolic model checking derives from using equations describing sets of states and sets of transitions to implicitly define the state space, thereby using significantly less memory than explicit representations that may enumerate the entire state space. Most symbolic model checkers utilize BDDs to accomplish this task, though other approaches are possible [158].

Boolean equations of the sort used to describe the transition systems plied by symbolic model checking are commonly represented in a variety of ways, such as truth tables and Binary Decision Trees (BDT), which are exponential in the size of the formulas they represent.

Definition 7

A **Binary Decision Tree (BDT)** *is a rooted, directed, acyclic graph (DAG) with vertices labeled by the n variables of the corresponding Boolean formula. Each variable node has exactly two children representing the two possible assignments to that variable, 0 and 1, labeled* low *and* high, *respectively. (Note that, except for the root, each node has exactly one parent.) A path through the BDT represents a valuation of the represented function given an assignment of values to the n variables; it starts at the root, follows the outgoing edge matching the assignment to the variable in the current vertex, and terminates in a vertex labeled either 0 or 1, corresponding to the value of the formula for that variable*

valuation. The BDT representing a formula over n variables has $2^n - 1$ variable vertices plus 2^n terminal vertices in $\{0, 1\}$, for a total size of $2^{n+1} - 1$ nodes.

A BDD is a refinement of a BDT. While in the worst case, a BDD is only roughly half the size[13] of its equivalent BDT, usually there is more significant improvement gained by the following construction.

Definition 8

*A **Binary Decision Diagram (BDD)** is a rooted, directed, acyclic graph (DAG) with internal vertices labeled by some sufficient subset of the n variables of the corresponding Boolean formula and exactly two terminal vertices, labeled 0 and 1. Each variable node has exactly two children representing the two possible assignments to that variable, 0 and 1, labeled* low *and* high, *respectively. (Note that, except for the root, each node may have any number of parents.) A path through the BDD represents a valuation of the represented function given an assignment of values to the n variables; it starts at the root, follows the outgoing edge matching the assignment to the variable in the current vertex, and terminates in the vertex corresponding to the value of the formula for that variable valuation.*

A BDD is considered ordered *if it follows a given total order of variables from root to leaves. Note that not all variables may appear in the tree or along any path in the tree; a variable only appears along a path if its value influences the value of the formula, given the assignments to previous variables along the path. If we are reasoning over multiple ordered BDDs (OBDDs), they are presumed to all follow the same variable ordering. A BDD is* reduced *if it contains no vertex v such that $low(v) = high(v)$ and no pair of vertices v_1 and v_2 such that the subgraphs rooted by v_1 and v_2 are isomorphic. That is, if there is a one-to-one function f such that for any vertex v_1' in the subgraph rooted at v_1, for some vertex v_2' in the subgraph rooted at v_2, $f(v_1') = v_2'$ implies that either v_1' and v_2' are both terminal vertices*

---

[13]Eliminating duplicate terminal nodes from a BDT of size $2^{n+1} - 1$ with 2^n terminal vertices leaves a BDD of size $2^n + 1$ nodes.

with the same value, or v'_1 and v'_2 are both non-terminal vertices with the same variable label and the property that $f(low(v'_1)) = low(v'_2)$ and $f(high(v'_1)) = high(v'_2)$, then either v_1 or v_2 will be eliminated. Note that each path realizes a unique set of variable assignments; each variable valuation corresponds to exactly one path. Furthermore, every vertex in the graph lies along at least one path; no part of the graph is unreachable.

In practice, any non-reduced BDD can be easily reduced by applying two reduction rules. The *redundancy elimination rule* removes any non-terminal node v_1 where $low(v_1) = high(v_1) = v_2$ and connects all incoming edges of v_1 directly to v_2 instead. The *isomorphism elimination rule* removes vertex v_1 wherever there exists a vertex v_2 such that $var(v_1) = var(v_2)$, $low(v_1) = low(v_2)$, and $high(v_1) = high(v_2)$ and redirects all incoming edges of v_1 directly to v_2. (We use $var(v)$ to denote the variable labeling vertex v.) Note that v_1 and v_2 may be either terminal or non-terminal vertices; this rule eliminates duplicate terminals as well as isomorphic subgraphs. A non-reduced BDD can be reduced in a single pass of the tree by careful application of these two rules in a bottom-up fashion. An example of reducing a Binary Decision Tree into a BDD is given in Figure 2.7. All three subfigures represent the formula $x_1 \lor x_2 \lor x_3$ with varying degrees of succinctness as each of the two reduction rules is applied.

(a) Binary Decision Tree

(b) isomorphism elimination rule

(c) redundancy elimination rule
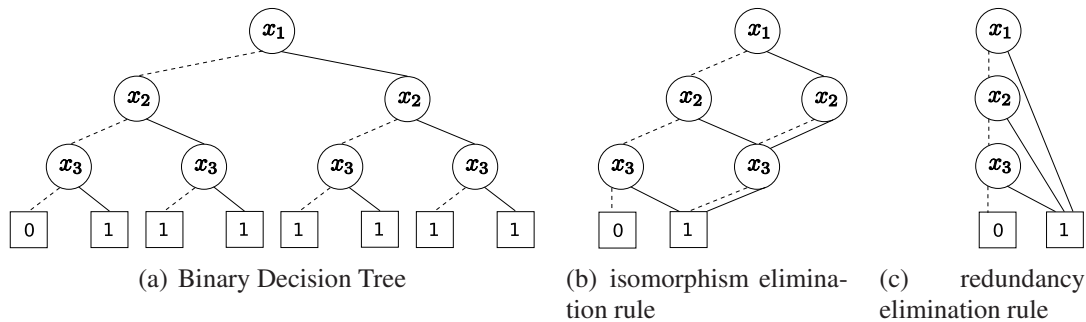
Figure 2.7 : Conversion of a Binary Decision Tree for $x_1 \lor x_2 \lor x_3$ into a Binary Decision Diagram.

When BDDs are both reduced and ordered (also called ROBDDs), they have many highly desirable properties. In this dissertation, we consider all BDDs to be both reduced and ordered; the algorithms used for symbolic model checking, and for efficient BDD manipulation in general, all maintain these properties. Reduced ordered BDDs are closed under the class of symbolic operations, as outlined in Chapter 2.9.2. These include APPLY and SATISFY-ONE, which form the basis for the symbolic model-checking algorithm.

Using BDDs to reason about Boolean formulas representing the state space offers many advantages over explicit automata representations:

- **Complexity (both time and space):** BDDs provide reasonably small representations for a large class of the most interesting, and most commonly-encountered, Boolean functions. For example, all symmetric functions, including the popular even and odd parity functions, are easily represented by BDDs where the number of vertices grows at most with the square of the number of arguments to the function. Moreover, since our BDDs are reduced and ordered, they have the property of minimality. This means if B is a BDD representing a function f, then for every BDD B' that also represents f and has the same variable ordering as B, $size(B) \leq size(B')$. Even better, we can frequently avoid enumerating the entire state space. Specifically, we can express tasks in several problem domains entirely in terms of graph algorithms over BDDs, describe automata in terms of sets of states and transitions, or use other techniques along those lines to express only relevant portions of the state space. Since all of the algorithms used for symbolic analysis of BDDs have complexities polynomial in the sizes of the input BDDs, the total computation remains tractable as long as the sizes of the input BDDs are reasonable. The sum of these time-and-space advantages allows us to use BDDs for solving problems that may scale beyond the limits of more traditional techniques such as case analysis or combinatorial search.

- **Canonicity:** There is a unique, canonical BDD representation for every Boolean function, given a total variable ordering. This property yields very simple algorithms for the most common BDD operations. Testing BDDs for equivalence just involves testing whether their graphs match exactly. Testing for satisfiability reduces to comparing the BDD graph to that of the constant function 0. Similarly, a function is a tautology iff its BDD is the constant function 1. Testing whether a function is independent of a variable x involves a simple check of whether its BDD contains any vertices labeled by x. Note that this is an important advantage over explicit automata representations since there is no canonical automaton representation for any class of automata.

- **Efficiency:** Besides the benefits that directly follow from canonicity, BDD representation enables efficient algorithms for many basic operations, including intersection, complementation, comparison, subtraction, projection, and testing for implication. The time complexity of any single operation is bounded by the product of the graph sizes of the functions being operated on or, for single-BDD operations like complementation, proportional to the size of the single function graph. In other words, the performance of directly manipulating BDDs in a sequence of operations degrades slowly, if at all.

- **Simplicity:** The BDD data structure and the set of algorithms for manipulating BDDs are conceptually and implementationally simple.

- **Generality:** BDDs can be used for reasoning about all kinds of finite systems, basically any problem that can be stated in terms of Boolean functions. They are not tied to any particular family of automata; they are just as useful for LTL symbolic model checking as they are for the extensions of LTL described in Chapter 2.2.2, for example.

The notion of using Binary Decision Diagrams to reason about Boolean functions was first introduced by Lee in 1959 [159] and later popularized by Akers who evinced the utility of BDDs for reasoning about large digital systems [160]. Fortune, Hopcroft, and Schmidt [161] restricted the ordering of the decision variables in the vertices of the graph, creating the canonical form of Ordered Binary Decision Diagrams (OBDDs) (though they called them B-schemes), noting the significance of variable ordering on graph size, and showing the ease of testing for functional equivalence. Bryant, who coined the term OBDDs, ameliorated ordered BDDs to symbolic analysis by demonstrating efficient algorithms for combining two functions with a binary operation and composing two functions [4, 162].

2.9.1   Representing Automata Using BDDs

Recall that $A_{M, \neg\varphi}$ is a Büchi Automaton with finite set of states Q, defined by assignments to the variables in the alphabet Σ, and transition relation $\delta : Q \times \Sigma \times Q$. We consider state $q_i \in Q$ as a tuple $\langle q_i, \sigma_0, \sigma_1, \ldots \rangle$ containing the name of the state and the assignments to each of the system variables, in order, in that state. In this dissertation, we presume that the set of system variables consists of Boolean-valued atomic propositions, which can each be represented by a single bit. However, this representation is easily extended to other data types by utilizing more bits to represent each system variable. For example, an integer variable with a range of $[0, 255]$ would be represented using 8 bits. (Unbounded variables cannot be encoded in this fashion as they cause the representation of $A_{M, \neg\varphi}$ to have an infinite number of states and thereby to be immune to the model-checking algorithm presented here. Infinite state model checking requires the use of alternative techniques, e.g. [163].) Choosing a Boolean encoding for an automaton is a bit of an art; there are several encoding variations from which to choose. When reasoning about a single automaton, for instance, there may be an advantage to encoding the state numbers in the representation along with the values of the variables in each state. For example, if our system model has 7 states with

5 system variables we need 4 bits to represent this range of state labels. The Boolean tuple representation of a state labeled "State 6" where all 5 system variables are false, would be $\langle 011000000 \rangle$. (The first 4 bits are the binary representation of the label 6 and the last 5 bits give the 5 false values of the system variables.) In practice, the state label is optional and can be included or not in the Boolean tuple for optimization purposes. This sort of encoding with state labels is not advantageous when reasoning about combinations of automata, as in symbolic model checking, as the state labels become meaningless. Also, we would rather avoid this kind of explicit enumeration of the states in the BDD representation since this does not offer a space-saving advantage over explicit-state model checking.

We represent symbolic automata using BDDs by encoding the transition relations directly. Basically, a transition is a pair of states: an origin state and a terminal state. A simple way to represent a transition from state q_i to state q_{i+1} is via the tuple $\langle q_i, q_{i+1} \rangle$, which is encoded using twice the number of bits as a single state. For instance, we can represent a transition from State 6 to State 7 in an example automaton with $|\Sigma| = 5$, presuming a specific variable ordering, as $\langle 0000001000 \rangle$. (The first 5 bits give the 5 false values of the system variables in State 6; the next 5 bits represent the values of the system variables in State 7, with the value of the second variable changed to $True$.)

Our BDD is constructed using $2|\Sigma|$ variables: two copies of each variable represent its value in the originating state and in the target state. The variables $\sigma_1, \sigma_2, \ldots, \sigma_n$ represent the values of the atomic propositions in the current state and the primed variables $\sigma'_1, \sigma'_2, \ldots, \sigma'_n$ represent the values of the atomic propositions in the next state. Finding an optimal BDD variable ordering is NP-complete and, for any constant $c > 1$, it is NP-hard to compute a variable ordering to make a BDD at most c times larger than optimal [164]. Though finding an optimal variable ordering is a tall order, a good rule of thumb is to group the most closely-related variables together in the ordering and some order involving interleaving the current and next time variants of the variable set together (as in

$\sigma_1, \sigma'_1, \sigma_2, \sigma'_2, \ldots, \sigma_n, \sigma'_n$) has been shown to have some nice advantages for model checking [33].

Alternative encodings include breaking a larger BDD representation into combinations of smaller BDDs to take advantage of natural logical separations and minimize the size of intermediate constructions [162]. Variations on the basic BDD structure can also be helpful. For example, shared BDDs, or BDDs with multiple root nodes, one for each function represented, save memory by overlapping BDD representations and adding a pair of hash tables for bookkeeping [165]. Oppositely, BDDs with expanded sets of terminal nodes (i.e. terminals other than 0 and 1), such as Multi-Terminal BDDs, offer unique advantages, especially for extensions of model checking that deal with uncertainty [166, 167]. Examining memory-saving refinements on the basic BDD structure that sustain or enhance the efficient manipulation of automata by reasoning over sets of states and sets of transitions is an ongoing area of research.

The Impact of Variable Ordering

The first decision we make when forming a BDD is the variable ordering. This is also the most important decision since the size of the graph representing any particular function may be highly sensitive to the chosen variable ordering. Many common functions, such as integer addition, have BDD representations that are linear for the best variable orderings and exponential for the worst. Unfortunately, computing an optimal ordering is an NP-complete problem. Even if we take the time to compute the optimal ordering, not all functions have reasonably-sized representations. For some systems, even a reasonably-sized BDD representation is too large to reason about practically, and for others, the BDD representation will grow exponentially with the size of the input function regardless of variable ordering [4]. Therefore, the best course we can take is to utilize a set of comparatively simple heuristics to choose an adequate ordering, avoiding orderings that would cause the

BDD to grow exponentially whenever possible. A classic example of a function with both linear and exponential BDD representations, depending on variable ordering, is displayed in Figure 2.8.

(a) variable order = $\{x_1, x_2, x_3, y_1, y_2, y_3\}$ (b) variable order = $\{x_1, y_1, x_2, y_2, x_3, y_3\}$
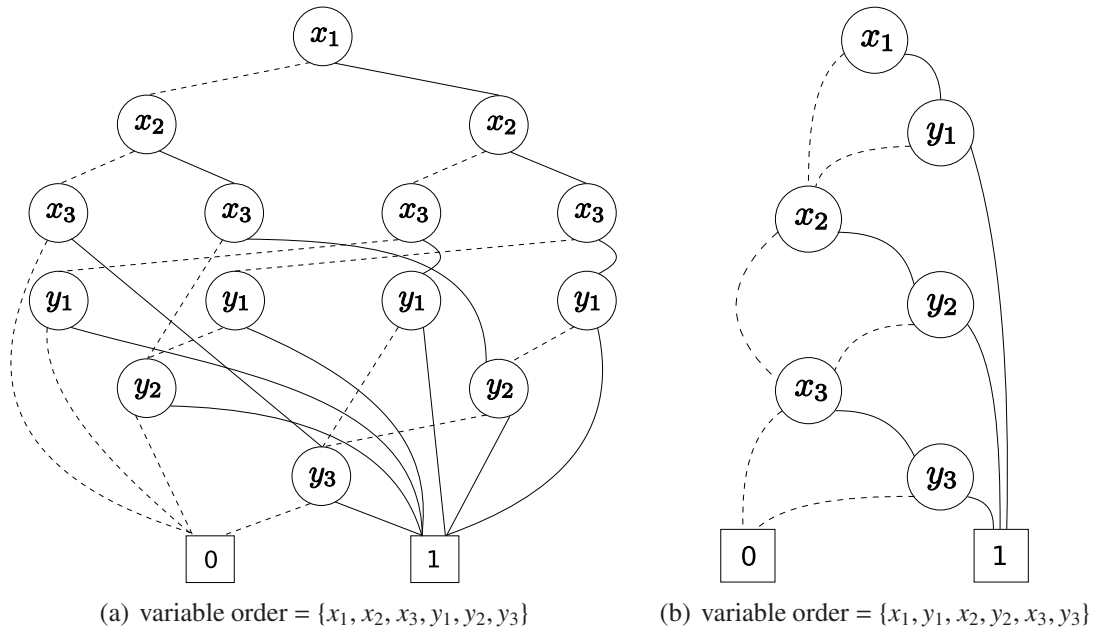
Figure 2.8 : BDDs for the function $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$.

2.9.2   BDD Operations

We introduce the algorithms underlying the symbolic model-checking procedure, as defined by Bryant [4] and reviewed by Andersen [3]. The full model-checking algorithm presented in Chapter 2.10 builds upon these basic operations.

If φ is purely propositional (i.e. contains no temporal operators) then we can easily construct a (reduced) BDD directly from $\neg\varphi$ using a straightforward algorithm like BUILD (shown in Figure 2.9), which simply loops through the variable ordering, recursively constructing each variable's low and high subtrees[14] [3]. Otherwise, we translate $\neg\varphi$ into a

---

[14]Unfortunately, this simple algorithm requires exponential (2^n) time.

```
algorithm BUILD(¬φ : propositional logic formula) : BDD
  // Build a reduced BDD for ¬φ

i : integer  // i indexes the variable order
n : integer  // n is the number of Boolean variables in Σ_BDD

//Recursive routine to implement BUILD
function BUILD-STEP(¬φ, i): vertex
begin
  if (i > n) then //if there are no variables left to add
    if (¬φ == false) then return 0
    else return 1
  //Recursive calls to build sub-trees
  else v0 = BUILD-STEP(¬φ[σ_i = 0], i + 1) //compute the low subtree
       v1 = BUILD-STEP(¬φ[σ_i = 1], i + 1) //compute the high subtree
       //new_vertex creates a new vertex only if v0 and v1 are different
       //  and the σ-labeled node with them as children
       //  does not already exist
       new_vertex(σ_i, v0, v1) //make/connect the new node
  return
end

return (BDD = BUILD-STEP(¬φ, 0))
```

Figure 2.9 : Pseudocode for the BUILD algorithm from [3].

symbolic automaton and then encode the result $A_{\neg\varphi}$ as a BDD, as described in Chapter 2.9.1.

Once we have built BDDs representing each of M and $A_{\neg\varphi}$, we need to compute their product, as described in Chapter 2.7. Let Σ_{BDD} be the set of BDD variables encoding Σ. The application of all binary operators over BDDs, including conjunction, disjunction, union, intersection, complementation (via *xor* with 1), and testing for implication, is implemented in a streamlined fashion utilizing a singular universal function. APPLY, shown in Figure 2.10, takes as input a binary operator and BDDs representing two functions f_1 and f_2, both over the alphabet $\Sigma_{BDD} = (\sigma_1, \ldots, \sigma_n)$ and produces a reduced graph representing the

function $f_1 < op > f_2$ defined as:

$$[f_1 < op > f_2](\sigma_1, \ldots, \sigma_n) = f_1(\sigma_1, \ldots, \sigma_n) < op > f_2(\sigma_1, \ldots, \sigma_n).$$

APPLY begins by comparing the root vertices of the two BDDs and proceeds downward. There are four cases. If vertices v_1 and v_2 are both terminal vertices, the result is simply the terminal vertex representing the result of $value(v_1) < op > value(v_2)$. If the variable labels of v_1 and v_2 are the same, we create a vertex in the result BDD with that variable label and operate on the pair of low subtrees, $low(v_1)$ and $low(v_2)$, and the pair of high subtrees, $high(v_1)$ and $high(v_2)$, to create the low and high subtrees of our new vertex, respectively. Otherwise, one of the variables, $var(v_1)$ or $var(v_2)$ is before the other in the total variable ordering. Note that this later vertex may or may not be a terminal vertex; either way, the algorithm is the same. If v_1 is the earlier vertex, then the function represented by the subtree with root v_2 is independent of the variable $var(v_1)$. (If this were not true, we would encounter the variable in both graphs.) In this case, we create a vertex in the result BDD labeled $var(v_1)$ and recur on the pair of subtrees $\{low(v_1), v_2\}$ to construct this new node's low subtree and on the pair of subtrees $\{high(v_1), v_2\}$ to construct this new node's high subtree. The last case, where $var(v_2)$ occurs before $var(v_1)$ in the total variable ordering, is symmetric. We presume `new_vertex()` is some function that creates a new vertex iff no isometric vertex exists, thus maintaining the reduced nature of the BDD under construction. Bryant [4] and Wegener and Sieling [168] describe implementation optimizations that yield a time complexity for this algorithm of $\mathcal{O}(|G_1||G_2|)$ where $|G_1|$ and $|G_2|$ are the sizes of the two graphs being operated upon.

Recall that finding a counterexample trace equates to finding a word in the language $\mathscr{L}(A_{M, \neg\varphi})$, which is essentially a satisfying assignment to the variables in Σ through time. The model checker uses a BDD-based fixpoint algorithm to find a *fair path* in the model-

```
algorithm APPLY(op : operator; v₁, v₂ : vertex): vertex
  // Evaluate v₁ op v₂


u : vertex   // u is the root vertex of the BDD representing v₁ op v₂
init(T)         // T is a hash table;
              // T(i, j) is either ∅ or
              //   the earlier computed result of Apply-step(i, j)


//Recursive routine to implement APPLY
function APPLY-STEP(v₁, v₂ : vertex): vertex
begin
  if (T(v₁, v₂) ≠ ∅) then return T(v₁, v₂) //this vertex pair has already
                                      // been evaluated
  else if (v₁ ∈ {0,1} and v₂ ∈ {0,1}) then //both v₁ and v₂ are
                                      // terminal vertices
    u = v₁ op v₂ //evaluate the simple Boolean expression
                // (This is the base case.)
  else if (var(v₁) == var(v₂)) then //if v₁ and v₂ are rooted
                                    // at the same variable
    //progress down both trees to the next variable in the ordering
    u = new_vertex(var(v₁), APPLY-STEP(low(v₁), low(v₂)),
          APPLY-STEP(high(v₁), high(v₂)))
  else if (var(v₁) < var(v₂)) then //if v₁ is first in the variable
                                   // ordering/nearer to root
    //compare the children of v₁ to v₂
    u = new_vertex(var(v₁), APPLY-STEP(low(v₁), v₂),
          APPLY-STEP(high(v₁), v₂))
  else if (var(v₁) > var(v₂)) then //if v₂ is first in the variable
                                   //ordering/nearer to root
    //compare v₂ to the children of v₂
    u = new_vertex(var(v₂), APPLY-STEP(v₁, low(v₂)),
          APPLY-STEP(v₁, high(v₂)))

  T(v₁, v₂) = u //store the result in T
  return u
end

u = APPLY-STEP(v₁, v₂)
return u
```

Figure 2.10 : The APPLY and APPLY-STEP algorithms from [4].

```
algorithm SATISFY-ONE(v: vertex, x: var_array) : Boolean
  //Find any satisfying assignment of the BDD rooted at v
  //  Return the counterexample in x

  if (v.value ==  0) then return false //unsatisfiable
  if (v.value ==  1) then return true  //satisfiable

  i = v.level //i is the level of vertex v,
              // also the place in the variable order
  x[i] = 0    //guess 0 for this vertex
  if SATISFY-ONE(v.low, x) then return true //satisfiable
  x[i] = 1    //backtrack if x[i] = 0 results in false
              // and guess 1 instead
  return SATISFY-ONE(v.high, x) //we know x[i] = 1 will be true
```

Figure 2.11 : Pseudocode for the SATISFY-ONE algorithm from [4].

automaton product [69]. The basic algorithm underlying this process is Bryant's SATISFY-ONE, shown in Figure 2.11 [4]. This algorithm is a simple recognition of the BDD principle that every non-terminal vertex has the terminal vertex labeled 1 as a descendant. Thus, a classic depth-first search with backtracking upon visiting the 0 terminal is guaranteed to find a satisfying path from the root to the terminal 1 in linear time.[15] Starting from the root, SATISFY-ONE arbitrarily guesses the low branch of each non-terminal vertex first, and stores a satisfying assignment (aka counterexample trace) of length $n = |\Sigma_{BDD}|$ in an array as it traverses the graph. It returns false if the function is unsatisfiable, and true otherwise.

Note that the reason we check for the existence of a single satisfying set and not for the whole set $S_{M, \neg\varphi}$ of satisfying sets is because enumerating the entire set requires time proportional to the length of the counterexamples times the number of elements in $S_{M, \neg\varphi}$. (For a propositional formula with $n = |\Sigma_{BDD}|$, the time complexity is $O(n \cdot |S_{M, \neg\varphi}|)$.) Considering the counterexample traces may be quite lengthy (i.e. n may be large), and there may be many of them, this is a highly inefficient check, unlikely to be performed in any reasonable

---

[15]One of the reasons maintaining a reduced BDD is important is that this algorithm can require exponential time if not [4].

timeframe. Furthermore, the utility of performing such a check is questionable: a single bug may generate any number of counterexamples. For verification purposes, where counterexamples may trace through many time steps, it is much more efficient to use the model checker to find a bug, then fix that bug before searching for additional counterexamples.

By changing the domain of the model-checking problem to reasoning over BDD operations from explicit manipulations of automata, we increase the size of the system we can reason about. Combining concepts from Boolean algebra and graph theory allows us to achieve such a high level of algorithmic efficiency that the performance of symbolic model checking using these techniques is limited mostly by the sizes of the BDDs involved. Table 2.2 compares the two methods for each step in the model-checking problem.

2.10 Checking for Counterexamples: Symbolically

We construct the automaton $A_{M,\,\neg\varphi}$ as the composition of the system automaton M and the automaton for the complemented LTL specification $A_{\neg\varphi}$. The model checking question, "does $M, q_0 \models \varphi$," then reduces to asking "is the language $\mathscr{L}(A_{M,\,\neg\varphi})$ empty?" The follow-up to this question is "if not, what word(s) does $A_{M,\,\neg\varphi}$ accept?" Any such word represents a computation of M that violates φ, constituting a *counterexample* to the check $M, q_0 \models \varphi$ and providing a valuable debugging tool for locating the bug. Thus, in practice, the model-checking step is performed in two parts: an emptiness check on the language $\mathscr{L}(A_{M,\,\neg\varphi})$ and the construction of the (preferably shortest) counterexample witness if $\mathscr{L}(A_{M,\,\neg\varphi}) \neq \emptyset$.

The earliest complete symbolic model-checking algorithm was designed in 1992 by Burch, Clarke, McMillan, Dill, and Hwang to take advantage of regularity in the state graph of the system and utilize the intuitive complexity of the state space rather than the explicit

---

[16]Here, the time complexity depends upon the size of the symbolic (BDD) representation in terms of the distances between states in the automaton graph, the number and arrangement of the strongly connected components in this graph, and the number of fairness conditions asserted.

Operation	Explicit Method	Time Complexity	Symbolic Method	Time Complexity
Translate M from a higher-level language	Construct a Büchi automaton	depends on M	Construct an ROBDD	$O(2^{|\Sigma \times \Sigma|})$
Translate φ from LTL to $A_{\neg\varphi}$	Construct a Büchi automaton	$2^{O(|\varphi|)}$	Construct an ROBDD	$O(2^{|el(\neg\varphi)|})$
Create $A_{M,\,\neg\varphi}$	Construct automaton for $\mathscr{L}(M) \cap \mathscr{L}(A_{\neg\varphi})$	$O(|M| \times |A_{\neg\varphi}|)$	Compute $(ROBDD_M \wedge ROBDD_{\neg\varphi})$	$O(|ROBDD_M| |ROBDD_{\neg\varphi}|) = O(2^{|\Sigma \times \Sigma|} 2^{|el(\neg\varphi)|})$
Check $A_{M,\,\neg\varphi}$ for nonemptiness	Search for an accepting path via iterative depth-first search of the strongly connected components	$O(|A_{M,\,\neg\varphi}|)$	Compute the fixpoint of the combined BDD with the spec $\mathcal{E}\square\ true$	$O(|BDD - representation|)$ [16]
Construct a counterexample	Compute an accepting cycle through the SCC graph	$O(|trace|)$	Find a path leading to the terminal node 1 by a depth-first traversal of $ROBDD_{M,\,\neg\varphi}$	Linear in the length of the counterexample found: $O(|trace|)$

Table 2.2 : **Table comparing explicit and symbolic model-checking algorithms.** Recall that $el(\neg\varphi)$ is the set of elementary formulas of φ as defined in Chapter 2.6. We presume the ROBDDs for M and $\neg\varphi$ are created using the appropriate variants of the BUILD algorithm [3]. The ROBDD for $A_{M,\,\neg\varphi}$ is created by an extension of the algorithm APPLY($\wedge$, ROBDD$_M$, ROBDD$_{\neg}\varphi$) [3], implementing the dynamic programming optimizations that result in lower time-complexities. Finally, the algorithm used for finding a counterexample given an ROBDD is based on ANYSAT [3]. Note that, for both explicit and symbolic model checking, multiple steps above are performed at once, using on-the-fly techniques, which increases the efficiency of the process and may avoid constructing all of $A_{M,\,\neg\varphi}$. We separate out the steps here for simplicity only.

size of the space [67]. We present an updated algorithm here, better suited to symbolic LTL model checking specifically [5], which takes full advantage of our automata-theoretic approach, allowing very naturally for extensions to the basic algorithm [39]. An issue with using the original symbolic model-checking algorithm, and later variants thereof, for LTL model checking is that they operated via a translation to the type of fair transition system also used for CTL model checking with fairness constraints [68], which itself involved a conversion to the μ-calculus [67, 25]. This method is indirect, complex, and does not account for the full expressibility of LTL such as *full fairness*, also called *strong fairness* or *compassion*. Recall from Chapter 2.5 that compassion extends justice to include the existence of sets of cyclically recurring system states. Compassion can be particularly useful when specifying special types of coordination, such as semaphores or synchronous communication protocols. In Chapter 2.5 we showed that compassion requirements can be included via specifications. We can also include compassion natively in the system model.

The classical automata-theoretic approach to model checking using justice corresponds to reasoning over generalized Büchi automata whereas adding compassion extends this to reasoning over Streett automata [169]. Streett automata are essentially the same as Büchi automata but with more general acceptance conditions, enabling the direct encoding of stronger notions of fairness [133]. Instead of specifying F, the set of final states that must be visited infinitely often in Büchi acceptance, Streett acceptance takes the form of a set of pairs of sets of states, (L_i, U_i) such that if a run visits L_i infinitely often, it must also visit U_i infinitely often [133]. The pairs of sets of states in Streett acceptance conditions correspond nicely to the definition of *generalized fairness* [170]. Of course, since Streett automata are expressively equivalent to Büchi automata and there is a straightforward conversion between the two [133], adding compassion at the algorithmic level does not substantially change our nonemptiness check [39].

Recall from Figure 2.5 that a counterexample witness takes the shape of a lasso: a finite

prefix starting from the initial state, q_0, and leading to an accepting cycle of the automaton $A_{M, \neg\varphi}$. For LTL model checking, fairness requirements are captured by the automata acceptance conditions, so each justice requirement and compassion requirement constitutes an additional acceptance condition. The challenge, then, is to find a reachable *fair cycle*, or a cycle that passes through at least one state satisfying each acceptance condition. Such a cycle is often referred to as a 'bad cycle' because it is a cycle on which all acceptance conditions (including fairness) are satisfied yet our specification, φ, is violated, which is certainly an undesirable outcome considering our goal is to prove that the system M never violates φ. We return as the counterexample, the list of states comprising the finite stem from q_0 to some state in this fair cycle, followed by a listing of the states visited in the cycle.

We want to find the shortest counterexample because that facilitates debugging and, in some cases, returning a shorter counterexample can save time and memory. However, this step in the model-checking process needs to be fast and finding the shortest counterexample is NP-complete [171], so we depend instead on reliable heuristics to efficiently find a (hopefully) short counterexample quickly.[17] Furthermore, we want to construct this counterexample directly from our symbolic automaton $A_{M, \neg\varphi}$ for maximum efficiency. For this reason, there has been a concentration on finding efficient symbolic cycle-detection algorithms. The first such effort, by Emerson and Lei [69], produced an algorithm that operates in quadratic time due to the presence of a doubly-nested fixpoint operator, but is still a standard to which all later algorithms are compared. All of these algorithms perform some iterative computation over the reachable states in the automaton $A_{M, \neg\varphi}$ to find paths to fair cycles. In order to accomplish this, it is necessary for us to treat this automaton as a graph and employ some classic techniques from graph theory.

---

[17]For an algorithm that always returns the shortest counterexample, see [172].

2.10.1 Automata as Graphs

Formally, a *predicate* over the set of states Q is any subset $P \subseteq Q$. A *binary relation* over Q is any set of pairs $R \subseteq Q \times Q$ such that, for predicates P_1 and P_2:

$$P_1 \times P_2 = \{(q_1, q_2) \in Q^2 | q_1 \in P_1, q_2 \in P_2\}.$$

Recall that a Büchi automaton can be viewed as a special case of a directed graph $G = (V, E)$ where V is the finite set of vertices, or the states of the automaton such that $V = Q$, and $E \subseteq V \times V$ is the set of edges, or the binary relation encapsulating the transitions of the automaton such that $(q, q') \in E$ whenever $\delta(q, \sigma) = q'$ for two states q and q' and any alphabet character $\sigma \in \Sigma$. A path through the graph that corresponds to a computation of the automaton, from some state q_j to state q_k is a sequence $< q_j, q_{j+1}, q_{j+2}, \ldots q_k >$ of vertices such that $(q_{i-1}, q_i) \in E$ for $i = j + 1, j + 2, \ldots, k$. Along such a path, we call vertex q_{i-1} a *predecessor* of state q_i and vertex q_{i+1} a *successor* of q_i. Note that a path must contain at least one edge but the presence of self-loops means that a state can be its own predecessor or successor. If there is a path from q_j to q_k, we say that q_k is *reachable* from q_j or that $q_j \rightsquigarrow q_k$. Let R represent the binary relation corresponding to the transition relation δ of $A_{M, \neg\varphi}$. We can extend this relationship for a predicate P and a relation R using pre- and post-composition as follows:

$$R \circ P = \{q \in Q | (q, q') \in R \text{ for some } q' \in P\}$$

$$P \circ R = \{q \in Q | (\hat{q}, q) \in R \text{ for } \hat{q} \in P\}$$

In other words, $R \circ P$ is the set of all predecessors of states in the set P. Conversely, $P \circ R$ is the set of all successors of states in the set P, or the set of all states reachable in a single transition from P in the graph of $A_{M, \neg\varphi}$, which we call $G_{A_{M, \neg\varphi}}$. We can employ

the $*$-operator to define $R^* \circ P$, the set of all states that can reach a P-state within a finite number (0 or more) steps. Similarly, we designate the set of all states reachable in a finite number of steps from a P-state as $P \circ R^*$. Expanded, we have:

$$R^* \circ P = P \cup R \circ P \cup R \circ (R \circ P) \cup R \circ (R \circ (R \circ P)) \cup \ldots$$

$$P \circ R^* = P \cup P \circ R \cup (P \circ R) \circ R \cup ((P \circ R) \circ R) \circ R \cup \ldots$$

Since $A_{M, \neg\varphi}$ is finite, it is clear that both $R^* \circ P$ and $P \circ R^*$ converge in a finite number of steps. Because $P \circ R^*$ consists of the set $\{q' \in Q | q \rightsquigarrow q'\}$, it is also called the *forward set* of state q. In the same vein, $R^* \circ P$ comprises the *backward set* of state q or $\{\hat{q} \in Q | \hat{q} \rightsquigarrow q\}$. We define the diameter d of a graph to be the length of the longest minimal path between any two states $(q_j, q_k \in G$ such that $q_j \rightsquigarrow q_k$. In other words, for all pairs of connected vertices $(q_j, q_k) \in G$, setting the distance from q_j to q_k to the smallest number of states on any path between them, d is the longest such distance in G.

A *connected component*, is a maximally connected subgraph such that two vertices are in the same connected component if and only if there is some path between them. Meanwhile, a *strongly connected component* (SCC) is a maximal subgraph such that every vertex in the component is reachable from every other vertex in the component. That is, for any strongly connected component S, for all vertices q_j, q_k such that $q_j \in S$ and $q_k \in S$, $q_j \rightsquigarrow q_k$ and $q_k \rightsquigarrow q_j$. An individual vertex, not connected to itself, comprises a *singular* or *trivial* SCC. A path $< q_j, q_{j+1}, q_{j+2}, \ldots q_k >$ forms a cycle if $q_j = q_k$ and the path contains at least one edge. Thus, a strongly connected component is essentially a set of vertices such that every pair of vertices in the set is contained in some cycle. Algorithms for finding SCCs in symbolic graphs take advantage of the following property:

Theorem 2.5

The intersection of a node's forward and backward sets is either empty or a strongly con-

nected component (SCC).

PROOF OF THEOREM 2.5

Let q be a vertex in directed graph G. Let $F(q)$ represent the forward set of q and $B(q)$ represent the backward set of q. First, presume q is in an SCC S. By definition of an SCC, all nodes are reachable from and can reach all other nodes in the SCC. Therefore, all of the nodes in the SCC containing q would have to be in both q's forward and backward sets, or $S \subseteq F(q) \cap B(q)$. Furthermore, no additional nodes not in the SCC containing q can be in the set $F(q) \cap B(q)$. Let q' be a node in $F(q) \cap B(q)$. Then $q \rightsquigarrow q'$ and $q' \rightsquigarrow q$. Since every node $\hat{q} \in S$ has a path to and can be reached from q, then it must be the case that q' and $\hat{q}$ are strongly connected by some path through q. In other words, $q' \rightsquigarrow q \rightsquigarrow \hat{q}$ and $\hat{q} \rightsquigarrow q \rightsquigarrow q'$. Therefore, we know that $F(q) \cap B(q) \subseteq S$. Together, we have $S \subseteq F(q) \cap B(q) \wedge F(q) \cap B(q) \subseteq S \rightarrow F(q) \cap B(q) = S$. It logically follows that the node q is not in an SCC iff $F(q) \cap B(q) = \emptyset$. ∎

A *terminal SCC* is an SCC with no edges leading out to nodes outside the SCC. Symmetrically, an *initial SCC* is one with no edges leading in from nodes outside the SCC. In Theorem 2.5 above, we take the intersection of node q's forward and backward sets because a node q in an SCC can still have a transition to another node outside that SCC so if the node q is in an SCC that is not terminal, there may be many more nodes in its forward set that are reachable from that SCC. Symmetrically, a node q in an SCC can still have a transition in from another node outside that SCC so if the node q is in an SCC that is not initial, there may be many more nodes in its backward set that can reach that SCC. All of the algorithms for symbolic cycle detection utilize some combination of computing forward sets (successors) while looking for initial SCCs, backward sets (predecessors) while looking for terminal SCCs, or some intermingling of both strategies.

Recall that the acceptance condition for a generalized Büchi automaton requires cycling

through a state in each set in F (the set of sets of states where each acceptance condition holds) infinitely often and that an accepting run of a generalized Büchi automaton resembles a lasso with a path from the start state to an accepting cycle. Thus, finding an accepting run of a generalized Büchi automaton comes down to finding a path to a strongly connected component in the graph of $A_{M, \neg\varphi}$, $G_{A_{M, \neg\varphi}}$, that contains at least one state satisfying each acceptance condition.

Strong Fairness   In LTL model checking, we may place these conditions on acceptance: that the counterexample found must be just and that it must be compassionate. For each justice requirement J in the set of justice requirements $\mathcal{J}$ over $A_{M, \neg\varphi}$, we define a set of J-states (states that satisfy J). We call a subgraph *just* if it contains a J-state for every justice requirement $J \in \mathcal{J}$. Similarly, for each compassion requirement over $A_{M, \neg\varphi}$, we define a pair of sets $(y, z) \in C$ that constitute that requirement where C is the set of all such pairs. We call a subgraph *compassionate* if, for every compassion requirement $(y, z) \in C$ the subgraph either contains a z-state or else does not contain any y-state. Combining these, we have that a subgraph is *fair* if it is a non-singular strongly connected component that is both just and compassionate, in that it intersects each accepting set of states in both $\mathcal{J}$ and C. Therefore, a counterexample lasso must be reachable from q_0 and cycle infinitely often through at least one state satisfying each acceptance condition, which involves visiting a minimum set of states satisfying any justice or compassion requirements we have asserted over $A_{M, \neg\varphi}$.

Theorem 2.6

[53] An infinite initialized path π is a computation of $A_{M, \neg\varphi}$ iff the set of states S that appear infinitely often in π comprise a fair SCC of the graph representing $A_{M, \neg\varphi}$, $G_{A_{M, \neg\varphi}}$.

Proof of Theorem 2.6

If-direction: π is a computation of $A_{M, \neg\varphi} \rightarrow S$ is a fair SCC in $G_{A_{M, \neg\varphi}}$.

Presume $A_{M, \neg\varphi}$ has a computation $\pi = \pi_0, \pi_1, \ldots$. By definition, π carves out an infinite path, starting in state q_0 that passes infinitely often through at least one state satisfying each acceptance condition (justice or compassion requirement) of $A_{M, \neg\varphi}$. Since $A_{M, \neg\varphi}$ is finite, π takes the shape of a lasso with a finite prefix of states starting from q_0 to a cycle containing all of the states π visits infinitely often. We define S to be the set of all states along that cycle. Then, S must be fair since π is an accepting run of $A_{M, \neg\varphi}$. Furthermore, S must be strongly connected since for all pairs of states $s, s' \in S$, both s and s' appear infinitely often in π, so $s \rightsquigarrow s'$ and $s' \rightsquigarrow s$.

Only-if direction: S is a fair SCC in $G_{A_{M, \neg\varphi}} \to \pi$ is a computation of $A_{M, \neg\varphi}$.

Presume S is a fair SCC in $G_{A_{M, \neg\varphi}}$. Consider some path π, originating in q_0 such that the set of states visited by π infinitely often is defined to be S. We can construct such a path by connecting every pair of states $s, s' \in S$ such that $(s, s') \in E$, which is equivalent to saying $\delta(s, \sigma) = s'$ for some character $\sigma \in \Sigma$. That is, we traverse every edge between states in S. In this way, we form the cycle such that there are infinitely many positions j such that $\pi_j = s$ and $\pi_{j+1} = s'$. By definition of a fair SCC, we know S intersects each accepting set of states in $G_{A_{M, \neg\varphi}}$. Therefore, by definition, π is a computation of $A_{M, \neg\varphi}$. ∎

2.10.2   Symbolic Methods for Graph Traversal

The standard algorithm for finding strongly connected components in a directed graph is depth-first search. Indeed, this is the algorithm used in explicit-state model checking for finding and returning counterexamples. However, in symbolic model checking we are not using an explicit automaton graph; our graph is instead encapsulated succinctly using BDDs. We have encoded the graph in terms of sets of states and sets of transitions, altering the problem of traversing the graph. Due to the nature of this encoding, depth-first approaches, while optimal for examining individual states, are not suitable for symbolic cy-

cle detection. Rather, we employ breadth-first, set-based cycle-detection algorithms better suited to searching over the characteristic function, S_h, as defined in Chapter 2.6.

Our goal is to locate a fair subgraph of $G_{A_{M,\neg\varphi}}$ because, as we know from Theorem 2.6, this will allow us to construct a counterexample, which is a computation of $A_{M,\neg\varphi}$. This search is an iterative process. Essentially, we will compute some part of the transitive closure of the transition relation of $A_{M,\neg\varphi}$, i.e. $R^* \circ P$ or $P \circ R^*$, preferably without incurring the computational expense of computing the entire transitive closure. To accomplish this, symbolic algorithms take advantage of meta-strategies for dealing with the set-based encoding of graphs such as intelligently partitioning $G_{A_{M,\neg\varphi}}$ and reasoning over the SCC quotient graph of $G_{A_{M,\neg\varphi}}$. The *SCC quotient graph* is a directed, acyclic graph $G_{quotient} = (V_{quotient}, E_{quotient})$ such that the set of vertices $V_{quotient}$ is the set of SCCs in $G_{A_{M,\neg\varphi}}$. Again, let R represent the binary relation corresponding to the transition relation δ of $A_{M,\neg\varphi}$. For two vertices $V_1, V_2 \in V_{quotient}$, there is an edge $(V_1, V_2) \in E_{quotient}$ iff $V_1 \neq V_2$ (so there are no self-loops) and $\exists v_1 \in V_1, v_2 \in V_2 : (v_1, v_2) \in R$. Restated, there is an edge in $G_{quotient}$ whenever there is an edge in $G_{A_{M,\neg\varphi}}$ from a state in one SCC to a state in a different SCC. Note that it is easy to prove that $G_{quotient}$ is acyclic since if there were a cycle in this graph, then all of the states in all of the SCCs on such a cycle would be reachable from each other, which contradicts their status as separate SCCs. $G_{quotient}$ defines a partial order over the SCCs in $G_{A_{M,\neg\varphi}}$ such that $v_1 \leq v_2$ for any states $v_1 \in V_1, v_2 \in V_2$ iff $v_1 \rightsquigarrow v_2$. Using $G_{quotient}$, we can identify the initial SCCs of $G_{A_{M,\neg\varphi}}$ as the sources of the graph and the terminal SCCs as the sinks. It is easier to reason about sets of predecessors and successors and partition the graph $G_{A_{M,\neg\varphi}}$ to limit our search. Symbolic algorithms utilize these types of tools to reduce the fair subgraph search problem to a smaller set of nodes than the entire graph $G_{A_{M,\neg\varphi}}$.

Whereas the time required for the depth-first exploration of the graph of $A_{M,\neg\varphi}$ performed in explicit-state model checking is directly dependent on the size of $A_{M,\neg\varphi}$, this

is not the case for symbolic model-checking algorithms, many of which are less efficient than that in the worst case. After all, symbolic algorithms derive their efficiency by representing compact characteristic functions instead of large numbers of individual states. In practice, the number of states in $A_{M, \neg \varphi}$ is not nearly as accurate a predictor of the time required for symbolic model checking as the graph diameter d, the length of the longest path in the SCC quotient graph, the number of justice and compassion requirements, the number of reachable SCCs (and how many of those are trivial), and the total number and size of the set of SCCs [173]. Basically, the number of states is frequently very large but some shapes of $A_{M, \neg \varphi}$ are much easier to model check than others. This is an advantage of utilizing symbolic algorithms since many very large graphs can be reduced to very succinct symbolic representations. The trade-off between the size of the representation of $A_{M, \neg \varphi}$ and the search complexity is necessary for practical verification. As the number of states in $A_{M, \neg \varphi}$ grows, considering every state individually quickly becomes infeasible so symbolic examination of the state space – while more difficult – is necessary to achieve scalability. Generally, the length of the counterexample found also depends heavily on the structure of the graph and the size of the symbolic representation.

There are many variations of this algorithm that have been created to provide better time-complexities, better performance times, and shorter counterexamples.[18] Optimizing fair cycle detection for symbolic model checking remains an active area of research [174, 172, 175, 176, 177, 173, 178, 179, 180].

There are two general classes of symbolic cycle-detection algorithms:

- **SCC-hull algorithms:** Most symbolic cycle-detection algorithms [69, 174, 166, 175, 5, 179] sequester an *SCC-hull*, or a set of states that contains all fair SCCs, without specifically enumerating the SCCs within. This set computation is not tight;

---

[18]Note that the problem of finding the shortest counterexample is NP-complete [171].

if there are fair SCCs in the graph, the SCC-hull algorithms may return extra states besides those in the fair SCCs.[19] Basically, these algorithms maintain an *approximation set*, or a conservative overapproximation of the SCCs. The approximation set is iteratively refined by *pruning*, or locating and removing states that cannot lead to a bad cycle utilizing the property that any state on a bad cycle must have a successor and a predecessor that are both on the same cycle, in the same SCC, and therefore in the approximation set. In the case that there are no fair SCCs, these algorithms return the empty set. These algorithms locate the SCC-hull using a fixpoint algorithm that either computes the set of all states with a path to a fair SCC [69], from a fair SCC [174], or some combination of both [175] since both of these sets include the states contained in a fair SCC, if one exists. Counterexamples are generated by searching for a path from the initial state q_0 to a state in a fair cycle then iterating through the set of acceptance conditions using breadth-first-search to find the shortest path to a state in the next fair set until the cycle is completed. Depending on the specific SCC-hull algorithm, varying amounts of additional work may be required to isolate the fair SCC to be returned in a counterexample.

- **Symbolic SCC-enumeration algorithms:** Alternatively, SCC-enumeration algorithms [172, 176, 177, 178] enumerate the SCCs of the state graph while taking advantage of the symbolic representation to avoid explicitly manipulating the states. Rather than extracting the (terminal or initial) SCCs from the set of all SCCs in a hull, these algorithms aim to compute reachability sets (either forward or backward) for fewer nodes by isolating SCCs sequentially. This is accomplished, for example, by recursively partitioning $G_{A_{M, \neg\varphi}}$ and iteratively applying reachability analysis to

---

[19]We can compile a tighter set of fair SCCs by directly computing the exact transitive closure of the transition relation rather than an overapproximation but this method is comparatively inefficient for reasoning over BDDs [173].

the subgraphs. The motivation is that many systems have only a limited number of fair SCCs so these algorithms can achieve a better worst case complexity than SCC-hull algorithms [178], but in practice their performance has been inferior [173]. Like with SCC-hull algorithms, these algorithms utilize some combination of forward and backward search, possibly in an interleaved manner. Pruning helps to reduce the number of trivial SCCs examined. After each SCC is identified, it is checked for fairness. The algorithm terminates as soon as the first fair SCC is located and may end up either enumerating all SCCs or paying additional computation cost for early elimination of unfair SCCs in the case that no fair SCC exists. Consequently, these algorithms tend to perform worse than SCC-hull algorithms when there are no fair SCCs or a large number of unfair SCCs present in $A_{M, \neg\varphi}$ [173]. Counterexamples are constructed similarly to, but possibly more efficiently than, SCC-hull algorithms since it is theoretically easier to locate a counterexample given that the enumerated fair SCC in question is tight. The length of the counterexample depends on which fair SCC is enumerated first, which is determined by the starting state of the algorithm, so optimizations include heuristics for choosing seed states in short counterexample traces.

2.10.3 SCC-hull Algorithm for Compassionate Model Checking

Here we present the full algorithm for symbolic LTL model checking, including for the sake of completeness, support for compassion at the algorithmic level, as opposed to adding it to the specification or allowing only weaker forms of fairness. All symbolic LTL model checkers support justice but not all support compassion. For example, at the time of this writing, NuSMV supports compassion [181] but CadenceSMV does not [182]. Currently, compassion requirements are rarely used in industrial practice. The algorithm discussed in this section was first published in 1998 by Kesten, Pnueli, and Raviv [5] and is exactly

the under-the-hood implementation in NuSMV today [181]. It is presented in a straight-forward manner using standard set theory since set operations on the languages accepted by finite automata translate transparently to logical operations on Boolean functions, as we discussed in Chapter 2.9.

We call an automaton *feasible* if it has at least one computation. The corollary to Theorem 2.6 is that $A_{M, \neg\varphi}$ is feasible iff $G_{A_{M, \neg\varphi}}$ contains a fair subgraph (a non-singular strongly connected component that is both just and compassionate). Therefore, we first check for the presence of a fair subgraph and determine the feasibility of our combined automaton. The model-checking problem reduces to $M \models \varphi$ iff $A_{M, \neg\varphi}$ is not feasible. If $A_{M, \neg\varphi}$ is not feasible, then our verification is complete; we can conclude that $M \models \varphi$. If $A_{M, \neg\varphi}$ is feasible, we will need to construct a (preferably short) counterexample.

Indeed, the algorithm for checking whether an automaton is feasible is essentially a specialization of the standard iterative algorithm for finding strongly connected components in a graph. The original cycle-detection algorithms for explicit-state model checking recursively mapped strongly connected subgraphs, computing closures under both successors and predecessors [43, 45]. Later it was proved that the requirement of bi-directional closure is too strong [5]. The algorithm, `FEASIBLE()`, presented in Figure 2.12, requires only closure under successor states while looking for initial components of the graph [5].[20] Proofs of soundness, completeness, and termination of this algorithm are provided in [5, 53].

The algorithm `FEASIBLE()` takes as input a synchronous parallel composition of the system and specification, which in our case is $A_{M, \neg\varphi}$. Recall that in this construction, $\neg\varphi$ is essentially serving as a tester of M, continuously monitoring the behavior of the system and making sure the system satisfies the desired property. Let $\|\psi\|$ denote the predicate consist-

---

[20]An equivalent algorithm would be to check for closure under predecessor states while looking for terminal components in the state-transition graph.

```
algorithm FEASIBLE(A_M, ¬φ): predicate // Check feasibility of A_M, ¬φ
new, old : predicate        //a predicate is a subset of
                              the set of states Q
R : relation                //a relation is a set of pairs of states

old = ∅                     //initialize old to the empty set
R  = ‖δ‖                    //R is the set of all state pairs in the
                              transition relation of A_M, ¬φ
new  = ‖q₀‖ ∘ R*            //new is the set of all states reachable from
                              a start state in A_M, ¬φ in a finite number
                              of steps
while (new ≠ old) do
begin
  old = new
  for each J ∈ 𝒥 do        //for each justice requirement J in the set 𝒥
    new  = (new ∩ ‖J‖) ∘ R* //new is reduced to the set of new states reach-
                              able in a finite number of transitions from
                              a state satisfying justice requirement J
  for each (y, z) ∈ C do   //for each compassion requirement (y, z)
                              in the set C
  begin
    new  = (new − ‖y‖) ∪ [(new ∩ ‖z‖) ∘ R*] //subtract from new all states
                                            where y holds and add to new all
                                            states reachable in a finite num-
                                            ber of steps from a z-state in new
    R = R ∩ (Q × new)      //subtract from R all transitions that don't
                             lead into a state in new
  end
  while (new ≠ new ∩ (new ∘ R)) do //while new is not comprised of the set
                                     of all successors of new states
    new = new ∩ (new ∘ R) //reduce new to states that have predecessors
                            in new
end
return (new)                //here, new should contain only fair
                             strongly connected components
```

Figure 2.12 : Pseudocode for the FEASIBLE algorithm from [5].

ing of all states that satisfy ψ, for some formula ψ, and $\|\delta\|$ denote the relation consisting of all state pairs $< q, q' >$ such that $\delta(q, \sigma) = q'$ for any alphabet character $\sigma \in \Sigma$. Our subroutine starts by computing the set of all successors of the initial state, q_0 in the predicate *new*. Because any run of $A_{M, \neg\varphi}$ will satisfy the initial condition and obey the transition relation but not necessarily satisfy the justice and compassion requirements, we need to check for this characteristic additionally. We loop through each justice and compassion requirement, subtracting states from *new* that are not reachable in a finite number of steps from states satisfying that requirement. Then we remove from *new* all states that do not also have a predecessor in *new* since such states cannot be part of a strongly connected component. We repeat this process until we reach a fixpoint; looping through every fairness requirement, checking for predecessors, and reducing *new* accordingly does not change the size of *new*. At this point, if *new* is empty, we return the emptyset and conclude that $A_{M, \neg\varphi}$ is not feasible and, therefore, $M \models \varphi$. If *new* is not empty, then it contains a fair strongly connected component of $G_{A_{M, \neg\varphi}}$ from which we must extract a counterexample witness.

To aid in our quest to efficiently find a short counterexample, if possible, we require a subroutine to find the shortest path between two states in $G_{A_{M, \neg\varphi}}$. Let λ represent the empty list. We use the $*$-operator to denote list *fusion* such that if $L_1 = (\ell_1, \ell_2, \ldots \ell_i)$ and $L_2 = (\ell_i, \ell_{i+1}, \ldots)$ are standard list data structures such that the last element of L_1 is the same as the first element of L_2, then $L_1 * L_2 = (\ell_1, \ell_2, \ldots \ell_i, \ell_{i+1}, \ldots)$ represents their concatenation, overlapping this shared element, or simply concatenating the lists if there is no overlap. We define the function *choose*(P) to consistently choose one element of a predicate P. Then we have the P A T H () subroutine in Figure 2.13.

We are now ready to present the full counterexample extraction algorithm, which we call W I T N E S S (), displayed in Figure 2.14. For completeness, we show the step of checking $A_{M, \neg\varphi}$ for feasibility in context. We define the function *last*(L_1) to return the last element in the list L_1.

```
// Compute shortest path from source to destination
function PATH(source, destination : predicate; R: relation) : list

start, f : predicate          //a predicate is a subset
                                of the set of states Q
L       : list               //a list is an ordered sequence of states
s       : state
start = source
L = λ                         //L is now the empty list
while (start ∩ destination == ∅) do
begin
  f = R ∘ destination           //start set f is the set of all predecessors
                                  of destination states
  while (start ∩ f == ∅) do //while we haven't found a path back to start
    f = R ∘ f                  //add all of the predecessors of the states
                                  currently in the set f
  s = choose(start ∩ f)        //pick one of the shortest path start states
  L = L * (s)                  //add this state to the end of the state list
  start = s ∘ R                //add all successors of s to the set start
end
return L * (choose(start ∩ destination)) //push the last state onto the end of
                                          the list and return the path
```

Figure 2.13 : Pseudocode of the PATH algorithm from [5].

After checking $A_{M, \neg\varphi}$ for feasibility, we exit if $M \models \varphi$. If not, we know that the set returned by FEASIBLE(), which we save in $final$, contains all of the states in fair SCCs that are reachable from q_0. We look at the set of successors of states in the set $final$. We pick a state from this set and iterate, replacing our chosen state s with one of its predecessors until the set of predecessors of s is a subset of the set of successors of s. Basically, we are attempting to move to an initial SCC in the quotient graph of $G_{A_{M, \neg\varphi}}$. Termination of this step is guaranteed by the finite nature of $G_{A_{M, \neg\varphi}}$ and the structure of the quotient graph. Next, we compute the precise SCC containing s, restrict our attention to transitions between states in this SCC, and utilize our shortest path algorithm to find the prefix of our counterexample, or the path from q_0 to our fair SCC. Then we construct the fair cycle through our SCC that satisfies all fairness requirements. In order to do this, we loop through each justice requirement, adding to our cycle a path to a state that satisfies that

```
algorithm WITNESS(A_M, ¬φ) : [list, list]
  // Extract a witness for A_M, ¬φ
final          : predicate
R              : relation
prefix, period : list
s              : state
final = FEASIBLE(A_M, ¬φ)                //the model-checking step
if (final == ∅) then return (λ, λ)       //∅ indicates that M, q₀ ⊨ φ
R = ‖δ‖ ∩ (final × Q)                    //R is the set of all transitions out
                                           of a state in a fair SCC in G_{A_M,¬φ}
s = choose (final)                       //pick one state from the set final
while (R* ∘ {s} − {s} ∘ R* ≠ ∅) do       //while the states from which we can
                                           reach s are not all states that can
                                           be reached from s

  s = choose(R* ∘ {s} − {s} ∘ R*)        //replace s with a predecessor that is
                                           not also a successor
final = R* ∘ {s} ∩ {s} ∘ R*              //compute the SCC containing s,
                                           i.e. the maximum set of states t such
                                           that (t ↝ s) ∧ (s ↝ t)

R = R ∩ (final × final)                  //R now contains only transitions
                                           within the SCC final

prefix = PATH(‖q₀‖, final, ‖δ‖)          //prefix is now a shortest path from an
                                           initial state to a state in final

period = (last(prefix))                  //period starts in the final state of
                                           prefix
for each J ∈ 𝒥 do                        //add in the justice requirements
  if (list−to−set(period) ∩ ‖J‖ == ∅) then //if there are no states in period
                                           satisfying justice requirement J
    period = period ∗ PATH({last(period)}, final ∩ ‖J‖, R)  //add to the end of
                                           period a shortest path to a state
                                           satisfying J
for each (y, z) ∈ C do                   //add in the compassion requirements
  if ((list−to−set(period) ∩ ‖z‖ == ∅) ∧ (final ∩ ‖y‖ ≠ ∅)) then
    //if this compassion requirement isn't satisfied
    period = period ∗ PATH({last(period)}, final ∩ ‖z‖, R) //add a z state
                                                     into period
period = period ∗ PATH({last(period)}, {last(prefix)}, R)

//finish period with a path to the end of prefix to form a complete loop
return(prefix, period)
```

Figure 2.14 : Pseudocode for the WITNESS algorithm from [5].

requirement if there is not one in the cycle already. We do the same for each compassion requirement. Finally, we complete the cycle with the shortest path back to the state we started the cycle from and we are ready to return our counterexample.

2.11   Discussion

While the complete algorithms presented here serve as the basis for all LTL model checking performed in industry, in practice, extensions to these algorithms are much more commonly used. We present the foundation upon which more specialized and scalable variations are built. Industrial applications frequently necessitate the use of extensions that provide additional scalability such as compositional verification [98], in which subunits of the system M and the interactions between these subunits are model checked separately, each in the manner presented here. Also, bounded model checking [158] depends upon a propositional SAT solver to replace the BDD operations of classical symbolic model checking and bounds the length of any possible counterexample to be less than some constant k. While this variation no longer provides the guarantee that no counterexample of any length exists, it vastly increases the size of systems we are able to verify, providing better guarantees of termination in a reasonable timeframe. Allowances for adding operators to LTL, as described in Chapter 2.2.2, are also implemented through adjustments to the algorithm for translating standard LTL.

Arguably the most pressing challenge in model checking today is scalability. We must make model-checking tools more efficient, in terms of the size and complexity of the models they can reason about and the time and space they require, in order to scale our verification ability to handle real-world safety-critical systems. Currently, LTL symbolic model checking is best used for systems whose state sets have short and easily manipulated descriptions [183]. However, we have previously mentioned that there are some functions

with unavoidably exponential symbolic representations.[21] In the future, we expect even more specialized variants of this algorithm to branch out from the common framework we have presented here in order to provide a greater array of options for practical LTL symbolic model checking of real-world systems, including those with characteristics we are currently unable to accommodate.

In the coming years, we expect to see more refined techniques for adding uncertainty in the model such as inaccurate sensor data or models of humans in the loop, abstractions for representing real-valued and infinite-range variables, algorithms that include probabilistic randomization, other extensions that allow probabilistic reasoning similar to rare-event simulation, and more tools for efficient model checking for extensions of LTL such as the industrial logics overviewed in Chapter 2.2.2.

---

[21] See [4] for a proof that a BDD representation of a function describing the outputs of an integer multiplier grows exponentially regardless of the variable ordering.

Chapter 3

Establishing Better Benchmarks

Before we introduce new constructions that we will argue perform better than the state-of-the-art, we address the questions of how the state of the art should be evaluated, and how we can fairly measure whether an alternative is "better." J.N. Hooker declared [184], "It is hard to make fair comparisons between algorithms and to assemble realistic test problems." Indeed, in the case of evaluating algorithms for LTL-to-automata, answering the question of what, precisely, is the state of the art and which algorithm(s) are best was, before the contributions of this thesis, a particularly daunting task. There are numerous papers introducing algorithms for constructing automata from LTL formulas and each one presents experimental results showing the superiority of that algorithm over other algorithms available at the time of publication. Most often, these works focused on measuring the size of the automata produced as a proxy for their "goodness." They assumed smaller automata would perform better, i.e. lead to faster model checking times, though there was no established rigorous basis for this assumption.

Previously, the best we could conclude was that, for every publicly available LTL-to-automaton algorithm, there exists some test in which that algorithm performs very well, where usually this meant that it produced smaller automata than other algorithms. In this thesis, we revisit the following fundamental questions:

1. **What should we be measuring?** What does it mean for one LTL-to-automaton algorithm to be "better" than another?

2. **How do we objectively measure it?** For every published LTL-to-automaton algo-

rithm, the authors include some experimental evaluation demonstrating that their algorithm is the best. How to we fairly evaluate all of these claims and pinpoint which algorithm(s) are actually best?

In this chapter, we address these two questions. In Section 3.1 we elaborate on the factors that we consider to constitute good benchmarks. In Sections 3.2, 3.3, and 3.4, we define three major classes of benchmarks consisting of *counter*, *pattern*, and *random* formulas, respectively, along with scalable models against which to check them defined in Section 3.5. We specialize these ideas to benchmark the performance of LTL-to-automaton algorithms in the context of model checking of safety properties in Section 3.6. Section 3.7 covers our new idea for taking into account correctness while benchmarking. We enumerate the performance aspects measured in our experimental evaluations and argue that our analysis provides an objective comparison of LTL-to-automaton algorithms in Section 3.8. More details on the code used for all of the benchmarks described here are available in Appendix A and online.

3.1   Importance of Good Benchmarks

We argue that a set of good benchmarks should necessarily address the following questions.

1. **What should we be measuring?** In this thesis, we fill a gap by addressing, in depth, the question: what does it mean for one algorithm to perform better than another? Many previous papers claimed that producing the smallest automata, in terms of number of states, number of transitions, or some combination of the two, makes an LTL-to-automaton algorithm superior. Intuitively, it makes sense that automaton size may be positively correlated with the time required to analyze that automaton, but no previous study objectively showed this was, in fact, the case. It was also not previously clear what times, precisely, we should be measuring. As we explained in Chap-

ter 2, there are many different stages in the execution of a model checker. Previous experimental results did not measure these separately or account for the sensitivity of the time required to complete each stage to influences unrelated to the automata being analyzed. We are the first to evaluate LTL-to-automaton algorithms based on performance time and not based on automaton size as a proxy for performance time.

- **For LTL satisfiability checking** we measure end-to-end performance time, i.e. given an automaton, how long does it take the model checker to perform the satisfiability check. Note that while this is the best measurement to answer the question of satisfiability checking performance, our conclusions about satisfiability checking performance may or may not be applicable to model checking performance.

- **For model checking** we measure the model checking time. Precisely, we measure the time for the model checker to perform the nonemptiness check, independently of any other preprocessing steps that occur beforehand. Note that an LTL-to-automaton algorithm that takes longer to create the automaton from the LTL formula, or creates an automaton that takes longer to compile, etc. may still perform "better" than one that requires less time for all of these preprocessing steps if the time required for the nonemptiness check is less. We are the first to evaluate LTL-to-automaton algorithms for model checking in this way. Our reasoning for this performance evaluation is that, while system models change frequently due to debugging and design changes, specifications change much less frequently. So, if we can can preprocess the specifications once and use them for model checking many times, the amortized cost is the least when the model checking time is minimized. Note that this capability is currently not available in the Spin model checker, which requires recompiling the specifica-

tions each time we rerun the model checker.

2. **How do we compare LTL-to-automaton algorithms fairly?** A good set of benchmarks should be challenging in diverse ways. Measuring the performance of LTL satisfiability checking enables us to benchmark the performance of LTL model checking tools, and, more specifically, of LTL translation tools. By using large LTL formulas, we offer challenging model-checking benchmarks to both the explicit and symbolic automaton domains. LTL formulas previously used for evaluating LTL translation tools are usually too small to offer challenging benchmarks. In part, this was because it was assumed that these formulas would be evaluated with respect to large system models. As modern safety-critical systems increase in size and complexity so too do the specifications describing safe system behaviors. Thus, studying satisfiability of large and complex LTL formulas is quite appropriate. Also, real specifications typically consist of many temporal properties, whose conjunction ought to be satisfiable. Scalability plays an important role in our experimental evaluations; our goal is to challenge LTL-to-automaton translations with large formulas and state spaces. We advocate the use of a wide variety of benchmark formulas that are designed to cover the range of possible formula constructions, address the question of scalability, address common patterns and combinations of operators that appear frequently in real-life specifications, and aid in checking the correctness of the generated automata.

Related Work Related software, called `lbtt`,[1] provides an LTL-to-Büchi explicit translator testbench and environment for basic profiling. The `lbtt` tool performs simple consistency checks on an explicit tool's output automata, accompanied by sample data when inconsistencies in these automata are detected [185]. Whereas the primary use of `lbtt` is

---

[1] `www.tcs.hut.fi/Software/lbtt/`

to assist developers of explicit LTL translators in debugging new tools or comparing a pair of tools, we compare performance with respect to LTL-to-automaton translations across a host of different tools, both explicit and symbolic.

3.2 Counter Formulas

Pre-translation rewriting is highly effective for many randomly-generated formulas, but ineffective for structured formulas [186, 136]. To measure performance on scalable, non-random formulas we tested LTL-to-automaton translation algorithms on formulas that describe n-bit binary counters with increasing values of n. These formulas are irreducible by pre-translation rewriting, uniquely satisfiable, and represent a predictably-sized state space. Whereas our measure of correctness for all other formulas is a conservative check that the automata produced for satisfiable LTL formulas are found by a model checker to be satisfiable, we check for precisely the unique counterexample for each counter formula. We introduce four constructions of binary counter formulas, varying two factors: number of variables and nesting of $\mathcal{X}$'s.

We can represent a binary counter using two variables: a *counter* variable and a *marker* variable to designate the beginning of each new counter value. Alternatively, we can use 3 variables, adding a variable to encode *carry* bits, which eliminates the need for $\mathcal{U}$-connectives in the formula. We can nest $\mathcal{X}$'s to provide more succinct formulas or express the formulas using a conjunction of unnested $\mathcal{X}$-sub-formulas.

Let b be an atomic proposition. Then a computation π over b is a word in $(2^{\{b\}})^\omega = \{0, 1\}^\omega$. By dividing π into blocks of length n, we can view π as a sequence of n-bit values, denoting the sequence of values assumed by an n-bit counter starting at 0, and incrementing successively by 1. To simplify the formulas, we represent each block $b_0, b_1, \ldots, b_{n-1}$ as having the most significant bit on the right and the least significant bit on the left. For

Figure 3.1 : Example of a 2-bit binary counter automaton (where a = marker; and b = counter).

example, for $n = 2$ the b blocks cycle through the values 00, 10, 01, and 11. Figure 3.1 pictures this 2-bit binary counter automaton. For technical convenience, we use an atomic proposition m to mark the blocks. That is, we intend the marker variable m to hold at point i precisely when $i = 0$ mod n.

For π to represent an n-bit counter, the following properties need to hold:

```
1) The marker consists of a repeated pattern of a 1
      followed by n-1 0's.
2) The first n bits are 0's.
3) If the least significant bit is 0, then it is 1 n steps later
      and the other bits do not change.
4) All of the bits before and including the first 0
      in an n-bit block flip their values in the next block;
      the other bits do not change.
```

For $n = 4$, these properties are captured by the conjunction of the following formulas:

```
1. (m) && ( [](m -> ((X(!m)) && (X(X(!m)))
```

```
                              && (X(X(X(!m))))

                              && (X(X(X(X(m))))))))
2. (!b) && (X(!b)) && (X(X(!b))) && (X(X(X(!b))))
3. []( (m && !b) ->

       ( X(X(X(X(b)))) &&

         X ( ( (!m) &&

            (b -> X(X(X(X(b))))) &&

            (!b -> X(X(X(X(!b))))) ) U m ) ) )
4. [] ( (m && b) ->

       ( X(X(X(X(!b)))) &&

         (X ( (b && !m && X(X(X(X(!b))))) U

             (m ||

              (!m && !b && X(X(X(X(b)))) &&

               X( ( !m && (b -> X(X(X(X(b))))) &&

                  (!b -> X(X(X(X(!b))))) ) U

                m ) ) ) ) ) ) )
```

Note that this encoding creates formulas of length $O(n^2)$. A more compact encoding results in formulas of length $O(n)$. For example, we can replace formula (2) above with:

```
2. ((!b) && X((!b) && X((!b) && X(!b))))
```

We can eliminate the use of $\mathcal{U}$-connectives in the formula by adding an atomic proposition c representing the carry bit. The required properties of an n-bit counter with carry are as follows:

```
1) The marker consists of a repeated pattern of a 1
   followed by n-1 0's.
2) The first n bits are 0's.
```

```
3) If m is 1 and b is 0 then c is 0 and n steps later b is 1.

4) If m is 1 and b is 1 then c is 1 and n steps later b is 0.

5) If there is no carry,

   then the next bit stays the same n steps later.

6) If there is a carry,

   flip the next bit n steps later and adjust the carry.
```

For $n = 4$, these properties are captured by the conjunction of the following formulas.

```
1. (m) && ( [](m -> ((X(!m)) && (X(X(!m)))

                              && (X(X(X(!m))))

                              && (X(X(X(X(m))))))))))

2. (!b) && (X(!b)) && (X(X(!b))) && (X(X(X(!b))))

3. [] ( (m && !b) -> (!c && X(X(X(X(b))))) )

4. [] ( (m && b) -> (c && X(X(X(X(!b))))) )

5. [] (!c && X(!m)) ->

       ( X(!c) && (X(b) -> X(X(X(X(X(b)))))) &&

                 (X(!b) -> X(X(X(X(X(!b))))))) )

6. [] (c -> ( ( X(!b) ->

                  ( X(!c) && X(X(X(X(X(!b)))))) ) ) &&

             ( X(c) && X(X(X(X(X(b))))) ) ))
```

The counterexample trace for a 4-bit counter with carry is given in Table 3.1. (The traces of *m* and *b* are, of course, the same as for counters without carry.)

3.3 Pattern Formulas

We further investigated the problem space by evaluating LTL-to-automaton algorithms using scalable pattern formulas. These formulas are comprised of regular patterns of combinations of temporal operators that were chosen either because they occur commonly in real

A 4-bit Binary Counter

m	1000	1000	1000	1000	1000	1000
b	0000	1000	0100	1100	0010	1010
c	0000	1000	0000	1100	0000	1000

m	1000	1000	1000	1000	1000	1000
b	0110	1110	0001	1001	0101	1101
c	0000	1110	0000	1000	0000	1100

m	1000	1000	1000	1000	1000	...
b	0011	1011	0111	1111	0000	...
c	0000	1000	0000	1111	0000	...

Table 3.1 : The counterexample trace for a 4-bit counter, including marker bit m, binary counter stream b, and carry bit c.

specifications or because they are known to be particularly difficult when they do occur. The scalable nature of this set of benchmarks makes them particularly nice for evaluating the scalability of the various algorithms under test.

For LTL satisfiability checking we evaluate the performance of LTL-to-automaton translation algorithms on temporally-complex formulas by testing them on eight patterns of scalable formulas defined by [64] plus one we defined and call R_2.

$$S(n) = \bigwedge_{i=1}^{n} \mathcal{G}p_i, \quad E(n) = \bigwedge_{i=1}^{n} \mathcal{F}p_i, \quad Q(n) = \bigwedge(\mathcal{F}p_i \vee \mathcal{G}p_{i+1}),$$

$$U(n) = (\dots(p_1 \, \mathcal{U} \, p_2) \, \mathcal{U} \, \dots) \, \mathcal{U} \, p_n, \quad U_2(n) = p_1 \, \mathcal{U} \, (p_2 \, \mathcal{U} \, (\dots p_{n-1} \, \mathcal{U} \, p_n)\dots),$$

$$C_1(n) = \bigvee_{i=1}^{n} \mathcal{GF}p_i, \quad C_2(n) = \bigwedge_{i=1}^{n} \mathcal{GF}p_i.$$

$$R(n) = \bigwedge_{i=1}^{n} (\mathcal{GF}p_i \vee \mathcal{FG}p_{i+1}), \quad R_2(n) = (\dots(p_1 \, \mathcal{R} \, p_2) \, \mathcal{R} \, \dots) \, \mathcal{R} \, p_n.$$

For model checking of safety properties we evaluate LTL-to-automaton translation algorithms using three challenging scalable safety formula patterns. Note that $\overline{E}$ is the

negation of the formula from [56, 64].

$$\overline{E}(n) = \neg \left(\bigwedge_{i=1}^{n} \mathcal{F} p_i \right),$$

$$X_1^*(n) = \bigvee_{i=1}^{n} (\Box \mathcal{X} p_i \vee \mathcal{X} \Box p_{i+1}),$$

$$M_2(n) = \Box \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} (\neg (p_i \wedge p_j)).$$

3.4   Random Formulas

Daniele, Guinchiglia, and Vardi previously addressed the problem of poor benchmarks for LTL-to-automaton algorithms in 1999 and set the original standard for LTL-to-automaton evaluation [135]. In order to cover as much of the problem space as uniformly possible, they defined a set of randomly-generated formulas varying:

- a number N of propositional variables

- a length L of the generated LTL formulas

- a number F of generated formulas

- a probability P of generating the temporal operators $\mathcal{U}$ and $\mathcal{R}$.

We repeat their test, expanding the ranges of L and F to account for the increase in processing power and LTL-to-automaton translation performance since 1999, as shown in Figure 3.2.

- sets of 500 randomly-generated formulas

- Scaling:

  1. number of variables [1, 2, 3]
  2. formula length [5, 10, . . . , 200]
  3. probability of occurrence of temporal operators [0.5]
     - Boolean operators: !, &&, ‖
     - Temporal operators: $\mathcal{X}$ (*next time*), $\mathcal{U}$ (*strong until*), $\mathcal{R}$ (implemented as !(!a $\mathcal{U}$!b))

Figure 3.2 : Composition of random formula benchmarks.

Thus, in order to cover as much of the problem space as uniformly as possible, we tested sets of 500 randomly-generated formulas varying the formula length and number of variables as in [135]. We randomly generated sets of 500 formulas varying the number of variables, N, from 1 to 3, and the length of the formula, L, from 5 up to 200. We set the probability of choosing a temporal operator $P = 0.5$ to create formulas with both a nontrivial temporal structure and a nontrivial Boolean structure. Other choices were decided uniformly. We note that the distribution of run times has a high variance and contains many outliers; this lead to our decision to generate large sets of random formulas and measure the median run times over each set. All formulas were generated prior to testing, so each LTL-to-automaton translation algorithm was run on the *same* formulas. While we made sure that, when generating a set of formulas of length L, every formula was exactly of length L and not *up to L*, we did find that the formulas were frequently reducible to simpler formulas via pre-translation rewriting. Conversely, subformulas of the form $\varphi \mathcal{R} \psi$ had to be expanded to $\neg(\neg\varphi \mathcal{U} \neg\psi)$ since most algorithms for LTL-to-automaton translation do not implement the $\mathcal{R}$ operator directly.

3.5   Scalable Universal Model

In all of our benchmark tests, the system model is a *universal* model, enumerating all possible traces over *Prop*. This universal model is defined differently in the explicit and symbolic domains.

3.5.1   Explicit Universal Model

For explicit-state model-checking benchmarks, our universal system model is a Promela (PROcess MEta LAnguage) program that explicitly enumerates all possible evaluations over *Prop* and employs nondeterministic choice to pick a new valuation at each time step. For example, when *Prop* = {*A*, *B*}, the Promela model is:

```
bool A,B;
/* define an active procedure
   to generate values for A and B */
active proctype generateValues()
{
  do
   :: atomic{ A = 0; B = 0; }
   :: atomic{ A = 0; B = 1; }
   :: atomic{ A = 1; B = 0; }
   :: atomic{ A = 1; B = 1; }
  od
}
```

We use the `atomic{}` construct to ensure that the Boolean variables change value in one unbreakable step. When combining formulas with this model, we also preceded each formula with an X-operator to skip Spin's assignment upon declaration and achieve nondeterministic variable assignments in the initial time steps of the test formulas. Note that the size of this model is exponential in the number of atomic propositions. It is also possible to construct a model that is linear in the number of variables like this:[2]

---

[2]We thank Martin De Wulf for asking this question.

```
bool A,B;
active proctype generateValues()
{
  do
   :: atomic{
      if
         :: true -> A = 0;
         :: true -> A = 1;
      fi;
      if
         :: true -> B = 0;
         :: true -> B = 1;
      fi;
   }
  od
}
```

For LTL satisfiability checking We use the exponentially-sized model for benchmarking explicit LTL-to-automaton translators since, in all of our counter and random formulas, there are never more than three variables. For these small numbers of variables, our (exponentially sized) model is simpler and contains fewer lines of code than the equivalent linearly sized model. When we did scale the number of variables for the pattern formula benchmarks, we kept the same model for consistency. We checked that the scalability of the universal model we chose did not affect our results. For example, all of the explicit LTL-to-automaton translation tool tests terminated early enough in our scaling sequence of pattern formula benchmarks that the size of the universal model was still reasonably small, usually 8 variables or less. (At 8 variables, our model has 300 lines of code, whereas the linearly sized model we show here has 38.) Furthermore, the timeouts and errors we encountered when comparing explicit-state LTL-to-automaton translation tools occurred in the LTL-to-automaton stage of the processing. All of the LTL-to-automaton implementations in our study spent considerably more time and memory on this stage, making the choice of universal Promela model in the counter and pattern formula benchmarks irrel-

evant: the LTL-to-automaton translation tools consistently terminated before the call to Spin to combine their automata with the Promela model. However, we must note that the choice of the format of the universal model may have significant impact for other studies, for example studies over benchmark formulas with more variables.

Checking Against Explicit State Universal Models   Each test was performed in two steps. First, we applied every LTL-to-automaton translation tool under test to the input LTL formula and ran them with the standard flags recommended by the tools' authors, plus any additional flag needed to specify that the output automaton should be in Promela. Second, each output automaton, in the form of a Promela `never` claim, was checked by Spin. (Spin `never` claims are descriptions of behaviors that should never happen.) In this role, Spin serves as a search engine for each of the LTL translation tools; it takes a `never` claim and checks it for nonemptiness in conjunction with an input model.[3] In practice, this means we call `spin -a` on the `never` claim and the universal model to compile these two files into a C program, which is then compiled using gcc and executed to complete the verification run.

3.5.2   Symbolic Universal Model

While very similar to each other, our symbolic universal models take slightly different forms depending on the input language for the symbolic model checking tool being used.

SMV   For the symbolic model checkers CadenceSMV and NuSMV, to check whether an LTL formula φ is satisfiable, we model check $\neg\varphi$ against a universal SMV model. For ex-

---

[3]An interesting alternative to Spin's nested depth-first search algorithm [60] would be to use SPOT's SCC-based search algorithm [187]. We did not do this because SPOT's model checking platform is not complete at this time but this option is considered for future work.

ample, if $\varphi = (X(a \; \mathcal{U} \; b))$, we provide the following inputs to NuSMV and CadenceSMV.[4]

NuSMV:

CadenceSMV:

```
MODULE main                      module main () {
  VAR
    a : boolean;                    a : boolean;
    b : boolean;                    b : boolean;
  LTLSPEC !(X(a=1 U b=1))          assert !(X(a U b));
  FAIRNESS                         FAIR TRUE;
    1                            }
```

For LTL satisfiability checking, SMV negates the specification, $\neg\varphi$, symbolically compiles φ into A_φ, and conjoins A_φ with the universal model. If the automaton is not empty, then SMV finds a fair path that satisfies the formula φ. In this way, SMV acts as both a symbolic compiler and a search engine.

SAL-SMC We also chose to evaluate SAL-SMC. We used a universal model similar to those for CadenceSMV and NuSMV. (In SAL-SMC, primes are used to indicate the values of variables in the next state.) For example, if $\varphi = (X(a \; \mathcal{U} \; b))$, we provide the following input to SAL-SMC.

```
temp: CONTEXT =
BEGIN

  main: MODULE =
  BEGIN
    OUTPUT
      a : boolean,
      b : boolean

    INITIALIZATION
```

---

[4]In our experiments we used FAIRNESS to guarantee that the model checker returns a representation of an infinite trace as counterexample.

```
      a IN {TRUE,FALSE};
      b IN {TRUE,FALSE};

   TRANSITION
    [  TRUE -->
         a' IN {TRUE,FALSE}; %next time a is in true or false
         b' IN {TRUE,FALSE}; %next time b is in true or false
    ]

 END; %MODULE

 formula: THEOREM main |- ((((G(F(TRUE)))))
                             => (NOT(((X (a U b)))))));

END %CONTEXT
```

For LTL satisfiability checking, SAL-SMC negates the specification, $\neg\varphi$, directly trans-lates φ into A_φ, and conjoins A_φ with the universal model. Like the SMVs, SAL-SMC then searches for a counterexample in the form of a path in the resulting model. There is not a separate command to ensure fairness in SAL-SMC models like those that appear in the SMV models above.[5] Therefore, we ensure SAL-SMC checks for an infinite counterexample by specifying our theorem as $\square \diamond (true) \rightarrow \neg\varphi$.

3.6   Benchmarks for Model Checking of Safety Properties

We can specialize our benchmarks for effectively evaluating the efficiency of algorithms for LTL-to-automata in the context of model checking of safety properties. We define two major categories of model-checking benchmarks employing the Universal Model: one in which we scale the model and one in which we scale the specifications.

---

[5]http://sal-wiki.csl.sri.com/index.php/FAQ#Does_SAL_have_constructs_ for_fairness.3F

3.6.1 Model-Scaling Benchmarks

We choose a set of 14 typical safety formulas, taken from common specifications and examples in the literature, listed in Table 3.2.

0	$\Box\neg bad$	"Something bad never happens."
1	$\Box(request \rightarrow \mathcal{X}grant)$	"Every request is immediately followed by a grant"
2	$\Box(\neg(p \wedge q))$	Mutual Exclusion: "p and q can never happen at the same time."
3	$\Box(p \rightarrow (\mathcal{X}\mathcal{X}\mathcal{X}q))$	"Always, p implies q will happen 3 time steps from now."
4^*	$\mathcal{X}((p \wedge q)\mathcal{R}r)$	"Condition r must stay on until buttons p and q are pressed at the same time."
5^*	$\mathcal{X}(\Box(p))$	slightly modified *intentionally safe* formula from [144]
6	$\Box(q \vee \mathcal{X}\Box p) \wedge \Box(r \vee \mathcal{X}\Box\neg p)$	*accidentally safe* formula from [144]
7^*	$\mathcal{X}([\Box(q \vee \Diamond\Box p) \wedge \Box(r \vee \Diamond\Box\neg p)] \vee \Box q \vee \Box r)$	slightly modified *pathologically safe* formula from [144]
8	$\Box(p \rightarrow (q \wedge \mathcal{X}q \wedge \mathcal{X}\mathcal{X}q))$	safety specification from [1]
9	$(((((p0\mathcal{R}(\neg p1))\mathcal{R}(\neg p2))\mathcal{R}(\neg p3))\mathcal{R}(\neg p4))\mathcal{R}(\neg p5))$	Sieve of Erathostenes [188, 189]
10	$(\Box((p0 \wedge \neg p1) \rightarrow (\Box\neg p1 \vee (\neg p1\mathcal{U}(p10 \wedge \neg p1)))))$	G.L. Peterson's algorithm for mutual exclusion algorithm [190, 191, 188, 192, 189]
11	$(\Box(\neg p0 \rightarrow ((\neg p1\mathcal{U}p0) \vee \Box\neg p1)))$	CORBA General Inter-Orb Protocol [193, 189]
12	$((\Box(p1 \rightarrow \Box(\neg p1 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box(p2 \rightarrow \Box(\neg p2 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box\neg p2 \vee (\neg p2\mathcal{U}p1)))$	GNU i-protocol, also called iprot [194, 192, 189]
13	$((\Box(p1 \rightarrow \Box(\neg p1 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box(p2 \rightarrow \Box(\neg p2 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box\neg p2 \vee (\neg p2\mathcal{U}p1)))$	Sliding Window protocol [195, 189]

Table 3.2 : Industrial safety formulas used in model-scaling benchmarks.

For each of the formulas in our set, we model check against a series of linearly-sized universal models, as described in Section 3.5.1, starting with the 10-variable model and

scaling up the number of variables in the model, thereby exponentially increasing its state space. Starred formulas are checked against universal models that set all variables to *true* first like this:

```
bool p,q;
active proctype generateValues()
{
  do
   :: atomic{
      if
         :: true -> p = 1;
         :: true -> p = 0;
      fi;
      if
         :: true -> q = 1;
         :: true -> q = 0;
      fi;
   }
  od
}
```

For our experiments, we name the specifications in Table 3.2, resulting in the following list of benchmarks: {SomethingBadNeverHappens, RequestGrant, MutualExclusion, Three-TimeSteps, ButtonPush, IntentionallySafe, AccidentallySafe, PathologicallySafeModified, Monitor, erathostenes-80-6, petersonN, giop3, iprot, sliding_window}.

3.6.2 Formula-Scaling Benchmarks

For our formula-scaling benchmarks, we model check each formula against a universal model with 30 variables and 1,073,741,824 states. We employ two types of formula-scaling benchmarks: pattern and random. We scale each of the formulas until model checking becomes unachievable within machine timeout.

pattern

We use three scalable pattern safety formulas. These patterns generate safety formulas that are easy to encode but difficult to model check, making them good benchmarks for evaluating the effects of different LTL-to-automaton encodings on model-checking performance. Note that $\overline{E}$ is the negation of the formula from Chapter 3.3 and [56, 64]. Again, starred formulas are checked against a universal model that sets all variables to *true* first.

$$\overline{E}(n) = \neg \left(\bigwedge_{i=1}^{n} \mathcal{F} p_i \right),$$

$$X_1^*(n) = \bigvee_{i=1}^{n} (\Box X p_i \vee X \Box p_{i+1}),$$

$$M_2(n) = \Box \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} (\neg(p_i \wedge p_j)).$$

random

We generated two sets each of 500 m-length safety specifications over n atomic propositions, for m in $\{5, 10, 15, 20, 25\}$ and n in $\{2..6\}$. (In total, there are 25,000 random formulas in these two benchmark sets, combined.) We set the probability of choosing a temporal operator $P = 0.5$. For the first set, we generate syntactic safety formulas, allowing negation only directly before atomic propositions and limiting the temporal operators to $\{X, G, R\}$. For the second set, rather than restricting ourselves to syntactic safety formulas, we generate each specification randomly over the full syntax of LTL. We then check if the generated specification represents a safety property using the SPOT command `ltl2tgba -O`, adding the specification to our test set if so and rejecting it if not.

3.7 Checking Correctness

Nearly always, when a new algorithm for LTL-to-automaton translation is created, the corresponding publication contains a theoretical proof of the algorithm's correctness. However, before our work, no previous evaluation had considered the correctness of the corresponding tool implementing the algorithm and no empirical evaluation had considered any measurement of correctness. While the one previous LTL-to-automata evaluation platform, `lbtt`, is helpful in testing whether an explicit LTL-to-automaton algorithm is *implemented consistently* via consistency checks that the output matches some expectation, it does not check whether an LTL-to-automaton algorithm *is correct* like we do. For example, `lbtt` uses only randomly-generated formulas; it does not use formulas with a known counterexample or minimum state count, like our counter formulas, which can provide invaluable sanity checks. Also `lbtt` can only perform checks on explicit-state automata in one particular format. It could not be used to compare the results of checking explicit SPOT automata to NuSMV and CadenceSMV symbolic automata, for example, which is how we uncovered the subtle bug that used to be in SPOT's translation at the time we conducted our experiments. (Following our study, the bug we found was corrected by the tool author, who also integrated our benchmarks into SPOT's test suite. Note that SPOT had previously been checked using `lbtt`, which did not locate the bug.)

Unfortunately, it is occasionally the case that some automata perform well because they are wrong, i.e. for the given LTL formula φ and the constructed Büchi automaton $A_\varphi = \langle Q, \Sigma, \delta, q_0, F \rangle$, $\mathscr{L}_\omega(A_\varphi)$ is *not* exactly $\models \varphi$. Constructing automata from LTL formulas is hard; it is easy to make mistakes in the implementation. Therefore, empirically examining the correctness of LTL-to-automaton implementations is important.

A major reason why previous work did not consider the evaluation of correctness is because it was not clear how correctness might be measured empirically. Certainly, for short

LTL formulas and small automata it is possible to check the correctness of the automata produced by hand, but most formulas used for verification in practice are too complex to analyze manually. Furthermore, a major reason for utilizing a model checker in verification is the automated nature of the tool; it is only necessary for system designers to learn the specification logic, not how the specifications are utilized inside the LTL-to-automaton translator and model checker after they are written.

We consider the automata produced by an LTL-to-automaton tool to be correct with very high confidence if:

1. In the context of LTL satisfiability checking, the back-end model checker always yields the one correct binary counter counterexample, for encodings of all four of our binary counter benchmark formulas, for all lengths of these formulas that can be tested before machine timeout.

2. The result of checking the automaton, "SAT" or "UNSAT" in the case of LTL satisfiability checking and "valid" or "violated" in the case of model checking, always matches the results of checking the automata produced by other LTL-to-automaton tools for the same formula, for all of the formulas in our *model-scaling* benchmarks and *pattern* and *random* formula benchmark sets.

3.8   Measurement and Analysis

Hooker also highlights the common fallacy introduced by competitive testing [184]: timing data tells us which algorithm is faster but not why. Indeed, timing data is often heavily influenced by factors including differences in the programming language used, whether it is an interpreted or compiled language, and the quality of interpreters or compilers available. Also, differences in the coding skills of the implementer, the level of tuning or algorithm specialization included, and the effort invested in creating very efficient code can dramat-

ically impact timing data despite being unrelated to the quality of the algorithm under implementation.

To address this problem, we measure LTL-to-automaton time separately from the time required to model check the produced automaton; this avoids the evils of competitive testing by isolating the influence of coding efficiency of the LTL-to-automaton translator from the performance of the automaton it produces. In effect, it allows us to judge the quality of the LTL-to-automaton algorithm independently from how the algorithm designers implemented it since the automata used as input to the back-end model checkers are all in the same language and format.

We also always examine the median times as well as the mean and standard deviation for all timing data measured in this thesis. We actually found the medians to be a better measure of performance times than the means in the experimental evaluations we conducted as, for all randomly-generated benchmarks, we encountered high standard deviations in the times measured and for all other benchmarks, we encountered higher-than-anticipated standard deviations. For example, benchmarking the same pattern formula against the same universal model multiple times produces different timing data, which may vary by several seconds when the benchmark formula is large. In the following chapters, we graph median timing data, which we found to provide a clearer picture of the trends in our data.

To address the widely-held belief that small automaton size may be positively correlated with fast performance, we measure the number of states and transitions of generated automata. When a counterexample is returned by the back-end model checker, we also record the length of the counterexample returned.

When we are comparing the influence on model checking times of algorithms for creating explicit automata from LTL formulas in Chapter 6, we further refine our measurements. The model checker Spin, which we used to model check the generated `never` claims we

are comparing, first translates each `never` claim from the input language Promela into C, then compiles the generated C code into binary, and finally executes the binary executable to perform the emptiness check. We measure each of these three stages of the model checker independently to further isolate the time required to actually perform the emptiness check from any biases that may be introduced, for example, from optimized code for translating more common Promela statements to C, or from inefficiencies in the native gcc compiler.

In summary, our analysis includes the following measurements, wherever applicable:

- measurement of the size or complexity of the benchmark being used, such as number of propositions in the formula, or any other distinguishing information

- number of formulas in the benchmark

- mean, standard deviation, and median number of states in the generated automaton (where measurable)

- mean, standard deviation, and median number of transitions in the generated automaton (where measurable)

- mean, standard deviation, and median LTL-to-automaton time

- for satisfiability checking: mean, standard deviation, and median back-end analysis time

- for model checking (with Spin):

  - mean, standard deviation, and median Promela→C translation time

  - mean, standard deviation, and median C→binary compilation time

  - mean, standard deviation, and median model checking time

- mean, standard deviation, and median counterexample length (over the formulas for which a counterexample is returned)

- total number and proportion of "valid" vs "violated" formulas

- total number and proportion of correct automata, as defined in Section 3.7

Chapter 4

LTL Satisfiability Checking

4.1 Introduction

The application of model-checking tools to complex systems involves a nontrivial step of creating a mathematical model of the system and translating the desired properties into a formal specification. When the model does not satisfy the specification, model-checking tools accompany this negative answer with a counterexample that points to an inconsistency between the system and the desired behaviors. It is often the case, however, that there is an error in the system model or an error in the formal specification. Such errors may not be detected when the answer of the model-checking tool is positive: while a positive answer does guarantee that the model satisfies the specification, the answer to the real question, namely, whether the system has the intended behavior, may be different.

Writing formal specifications is a difficult task, which is prone to error just as implementation development is error prone. However, formal verification tools offer little help in debugging specifications other than standard vacuity checking. Clearly, if a formal property is valid, then this is certainly due to an error. Similarly, if a formal property is *unsatisfiable*, that is, true in *no* model, then this is also certainly due to an error. Even if each individual property written by the specifier is satisfiable, their conjunction may very well be unsatisfiable. Recall that a logical formula φ is valid iff its negation $\neg\varphi$ is not satisfiable. Thus, as a necessary sanity check for debugging a specification, model-checking tools should ensure that both the specification φ and its negation $\neg\varphi$ are satisfiable. (For a different approach to debugging specifications, see [121].)

Recall from Chapter 2.3 that we can relate LTL satisfiability checking to LTL model checking. Suppose we have a *universal model M* that generates all computations over its atomic propositions; that is, we have that $L_\omega(M) = (2^{Prop})^\omega$. For an LTL formula φ, we now have that $M \not\models \neg\varphi$ if and only if φ is satisfiable. Recall that to check whether $M \models \neg\varphi$ we first negate the specification ($\neg\neg\varphi = \varphi$), then the model checker searches for an accepting run of the synchronous parallel composition of M and φ and returns a counterexample if one exists. Thus, φ is satisfiable precisely when the model checker finds a counterexample and that counterexample represents a satisfying computation of φ. We argue that it is easy and necessary to add a satisfiability-checking feature to LTL model-checking tools.

There has been extensive research over the past decade into explicit translation of LTL to automata [130, 135, 186, 196, 141, 138, 137, 134, 136, 139, 63], but it is difficult to get a clear sense of the state of the art from a review of the literature. Measuring the performance of LTL satisfiability checking enables us to benchmark the performance of LTL model-checking tools, and, more specifically, of LTL-to-automaton translation tools.

We report here on an experimental investigation of LTL satisfiability checking via a reduction to model checking. By using large LTL formulas, we offer challenging benchmarks to both explicit and symbolic model checkers. In the symbolic domain, we used CadenceSMV, NuSMV, and SAL-SMC both to translate our benchmark formulas into symbolic automata and to search for a satisfying counterexample. In the explicit domain, we used Spin as the back-end search engine, and we tested essentially all publicly available LTL-to-automaton translation tools. We used a wide variety of benchmark formulas as described in Chapter 3, either generated randomly, as in [135], or using a scalable pattern (e.g., $\bigwedge_{i=1}^{n} p_i$). LTL formulas typically used for evaluating LTL translation tools are usually too small to offer challenging benchmarks. Note that real specifications typically consist of many temporal properties, whose conjunction ought to be satisfiable. Thus, studying satisfiability of large LTL formulas is quite appropriate.

Our experiments resulted in two major findings. First, most LTL translation tools are research prototypes and cannot be considered industrial quality tools. Many of them are written in scripting languages such as Perl or Python, which has drastic negative impact on their performance. Furthermore, as we increase the size and difficulty of benchmark LTL formulas these tools generally degrade gracelessly, often yielding incorrect results with no warning. Among all of the explicit LTL-to-automaton tools we tested, only SPOT can be considered an industrial quality tool. Second, when it comes to LTL satisfiability checking, the symbolic approach is clearly superior to the explicit approach. Even SPOT, the best explicit LTL translator in our experiments, was rarely able to compete effectively against the symbolic tools. This result is consistent with the comparison of explicit and symbolic approaches to modal satisfiability [197, 198], but is somewhat surprising in the context of LTL satisfiability in view of [199].

Related software, called `lbtt`,[1] provides an explicit LTL-to-Büchi automaton translator testbench and an environment for basic profiling. The `lbtt` tool performs simple consistency checks on an explicit tool's output automata, accompanied by sample data when inconsistencies in these automata are detected [185]. While `lbtt` is useful for assisting developers of explicit LTL translators in debugging new tools or comparing a pair of tools, its functionality is not comparable to an analysis using our rigorous benchmark suite. For example we compare explicit and symbolic tools to each other and measure aspects that `lbtt` cannot, such as correctness, as explained in Chapter 3.7.

This chapter is structured as follows. In Section 4.2, we describe the tools studied in our investigation. We define our experimental method in Section 4.3, and detail our four major findings in Sections 4.4, 4.5, 4.6, and 4.7. We conclude with a discussion in Section 4.8.

---

[1]`www.tcs.hut.fi/Software/lbtt/`

4.2   Tools for LTL-to-Automata

In total, we tested eleven LTL compilation algorithms from nine research tools. To offer a broad, objective picture of the current state of the art, we tested the algorithms against several different sequences of benchmarks and compared, where appropriate, the size of generated automata in terms of numbers of states and transitions, translation time, model-analysis time, and correctness of the output.

4.2.1   Explicit Tools

The explicit LTL model checker Spin [59] accepts either LTL properties, which are translated internally into Büchi automata, or Büchi automata for complemented properties (called "`never` claims"). The states and transitions of these Büchi automata are explicitly represented in `never` claims. The model checking algorithm used by Spin appears in Chapter 2.8. We tested Spin with Promela (PROcess MEta LAnguage) `never` claims produced by several LTL translation algorithms. (As Spin's built-in translator is dominated by TMP, we do not show results for this translator.) The algorithms studied here represent all tools publicly available in 2006, as described in Table 4.1.

We provide here short descriptions of the tools and their algorithms, detailing aspects that may account for our results. We also note that aspects of implementation including programming language, memory management, and attention to efficiency, seem to have significant effects on tool performance.

Classical Algorithms   Following [44], the first optimized LTL translation algorithm was described in [134]. The basic optimization ideas were: (1) generate states by demand only, (2) use node labels rather than edge labels to simplify translation to Promela, and (3) use a *generalized Büchi* acceptance condition so eventualities can be handled one at a time. The resulting generalized Büchi automaton (GBA) is then "degeneralized" or translated

Explicit Automata Construction Tools
LTL2AUT . . . . (Daniele–Guinchiglia–Vardi)
Implementations (Java, Perl) . . . . LTL2Buchi, Wring
LTL2BA (C) . . . . (Oddoux–Gastin)
LTL2Buchi (Java) . . . . (Giannakopoulou–Lerda)
LTL → NBA (Python) . . . . (Fritz–Teegen)
Modella (C) . . . . (Sebastiani–Tonetta)
SPOT (C++) . . . .
. . . . (Duret-Lutz–Poitrenaud–Rebiha–Baarir–Martinez)
TMP (SML of NJ) . . . . (Etessami)
Wring (Perl) . . . . (Somenzi–Bloem)

Table 4.1 : List of tools for translating LTL formulas into explicit automata.

to a BA. **LTL2AUT** improved further on this approach by using lightweight propositional reasoning to generate fewer states [135]. We tested two implementations of LTL2AUT, one included in the Java-based LTL2Buchi tool and one included in the Perl-based Wring tool.

TMP[2] [186] and **Wring**[3] [136] each extend LTL2AUT with three kinds of additional optimizations. First, in the *pre-translation optimization*, the input formula is simplified using Negation Normal Form (NNF) and extensive sets of rewrite rules that differ between the two tools; for example TMP adds rules for left-append and suffix closure. Second, *mid-translation optimization*s tighten the LTL-to-automaton translation algorithms. TMP optimizes an LTL-to-GBA-to-BA translation, while Wring performs an LTL-to-GBA translation utilizing Boolean optimizations for finding minimally-sized covers. Third, the resulting automata are minimized further during *post-translation optimization*. TMP mini-

---

[2]We used the binary distribution called `run_delayed_trans_06_ compilation.x86-linux`. `www.bell-labs.com/project/TMP/`

[3]Version 1.1.0, June 21, 2001. `www.ist.tugraz.at/staff/bloem/wring.html`

mizes the resulting BA by simplifying edge terms, removing "never accepting" nodes and fixed-formula balls, and applying a fair simulation reduction variant based on partial-orders produced by iterative color refinement. Wring uses forward and backward simulation to minimize transition- and state-counts, respectively, merges states, and performs fair set reduction via strongly connected components. Wring halts translation with a GBA, which we had to degeneralize.

LTL2Buchi[4] [138] optimizes the LTL2AUT algorithm by initially generating a transition-based generalized Büchi automaton (TGBA) rather than a node-labeled BA, to allow for more compaction based on equivalence classes, contradictions, and redundancies in the state space. Special attention to efficiency is given during the ensuing translation to a node-labeled BA. The algorithm incorporates the formula rewriting and BA-reduction optimizations of TMP and Wring, producing automata with less than or equal to the number of states and fewer transitions.

Modella[5] focuses on minimizing the *nondeterminism* of the property automaton in an effort to minimize the size of the product of the property and system model automata during verification [139]. If the property automaton is deterministic, then the number of states in the product automaton will be at most the number of states in the system model. Thus, reducing nondeterminism is a desirable goal. This is accomplished using *semantic branching*, or branching on truth assignments, rather than the *syntactic branching* of LTL2AUT. Modella also postpones branching when possible.

Alternating Automata Tools Instead of the direct translation approach of [44], an alternative approach, based on *alternating automata*, was proposed in [200]. In this approach,

---

[4]Original Version distributed from `http://javapathfinder.sourceforge.net/`; description: `http://ti.arc.nasa.gov/profile/dimitra/projects-tools/\#LTL2Buchi`

[5]Version 1.5.8.1. `http://www.science.unitn.it/~stonetta/modella.html`

the LTL formula is first translated into an alternating Büchi automaton, which is then translated into a nondeterministic Büchi automaton.

LTL2BA[6] [137] first translates the input formula into a *very weak* alternating automaton (VWAA). It then uses various heuristics to minimize the VWAA, before translating it into a GBA. The GBA in turn is minimized before being translated into a BA, and finally the BA is minimized further. Thus, the algorithm's central focus is on optimization of intermediate representations through iterative simplifications and on-the-fly constructions.

LTL→NBA[7] follows a similar approach to that of LTL2 BA [196]. Unlike the heuristic minimization of the intermediate VWAA used in LTL2BA, LTL→NBA uses a game-theoretic minimization based on utilizing a delayed simulation relation for on-the-fly simplifications. The novel contribution is that that simulation relation is computed from the VWAA, which is linear in the size of the input LTL formula, *before* the exponential blow-up incurred by the translation to a GBA. The simulation relation is then used to optimize this translation.

Back to Classics **SPOT**[8] is the most recently developed LTL-to-Büchi optimized translation tool [140]. It does not use alternating automata, but borrows ideas from all the tools described above, including reduction techniques, the use of TGBAs, minimizing nondeterminism, and on-the-fly constructions. It adds two important optimizations: (1) unlike all other tools, it uses pre-branching states, rather than post-branching states (as introduced in [130]), and (2) it uses BDDs [4] for propositional reasoning.

---

[6]Version 1.0; October 2001. `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php`

[7]This original version is a prototype. `http://www.ti.informatik.uni-kiel.de/~fritz/` `;download:http://www.ti.informatik.uni-kiel.de/~fritz/LTL-NBA.zip`

[8]Version 0.3. `http://spot.lip6.fr/wiki/SpotWiki`

4.2.2 Symbolic Tools

Symbolic model checkers describe both the system model and property automaton symbolically: states are viewed as truth assignments to Boolean state variables and the transition relation is defined as a conjunction of Boolean constraints on pairs of current and next states [67]. The model checker uses a BDD-based fixpoint algorithm [69, 5] to find a *fair path* in the model-automaton product, as described in Chapter 2.10.

CadenceSMV[9] [14] and **NuSMV**[10] [20] both evolved from the original Symbolic Model Verifier developed at CMU [201]. Both tools support LTL model checking via the translation of LTL to symbolic automata with FAIRNESS constraints, as described in [68]. FAIRNESS constraints specify sets of states that must occur infinitely often in any path. They are necessary to ensure that the subformula ψ holds in some time step for specifications of the form $\varphi \, \mathcal{U} \, \psi$ and $\Diamond \psi$. CadenceSMV additionally implements heuristics that attempt to reduce LTL model checking to CTL model checking in some cases [202].

SAL[11] (Symbolic Analysis Laboratory), developed at SRI, is a suite of tools combining a rich expression language with a host of tools for several forms of mechanized formal analysis of state machines [203]. SAL-SMC (Symbolic Model Checker) uses LTL as its primary assertion language and directly translates LTL assertions into Büchi automata, which are then represented, optimized, and analyzed using BDDs. SAL-SMC also employs an extensive set of optimizations during preprocessing and compilation, including partial evaluation, common subexpression elimination, slicing, compiling arithmetic values and operators into bit vectors and binary "circuits," as well as optimizations during the direct translation of LTL assertions into Büchi automata [76].

---

[9]Release 10-11-02p1. `http://www.kenmcmil.com/smv.html`

[10]Version 2.4.3-zchaff. `http://nusmv.irst.itc.it/`

[11]Version 2.4. `http://sal.csl.sri.com`

4.3   Experimental Methods

4.3.1   Performance Evaluation

We ran all tests in the fall of 2006 on Ada, a Rice University Cray XD1 cluster.[12] Ada is comprised of 158 nodes with 4 processors (cores) per node for a total of 632 CPUs in pairs of dual core 2.2 GHz AMD Opteron processors with 1 MB L2 cache. There are 2 GB of memory per core or a total of 8 GB of RAM per node. The operating system is SuSE Linux 9.0 with the 2.6.5 kernel. Each of our tests was run with exclusive access to one node and was considered to time out after 4 hours of run time. We measured all timing data using the Unix `time` command.

4.3.2   Input Formulas

We benchmarked the tools against three types of scalable formulas: counter formulas (Chapter 3.2), pattern formulas (Chapter 3.3), and random formulas (Chapter 3.4). Scalability played an important role in our experiments, since the goal was to challenge the tools with large formulas and state spaces. All tools were applied to the same formulas and the results (satisfiable or unsatisfiable) were compared. The symbolic tools, which were always in agreement, were considered as reference tools for checking correctness as described in Chapter 3.7.

4.3.3   Explicit Tools

Each test was performed in two steps. First, we applied the translation tools to the input LTL formula and ran them with the standard flags recommended by the tools' authors, plus any additional flag needed to specify that the output automaton should be in Promela. Second, each output automaton in the form of a Promela `never` claim was checked by

---

[12]`http://rcsg.rice.edu/ada/`

Spin. (Note that we do not negate φ before creating a corresponding `never` claim; Spin `never` claims are descriptions of behaviors that should never happen so this test checks for counterexamples satisfying φ.) In this role, Spin serves as a search engine for each of the LTL translation tools; it takes a `never` claim and checks it for nonemptiness in conjunction with an input model.[13] In practice, this means we call `spin -a` on the `never` claim and the universal model to compile these two files into a C program, which is then compiled using gcc and executed to complete the verification run.

In all tests, the model was a *universal* Promela program, enumerating all possible traces over *Prop*. We use the (exponentially-sized) explicit universal Promela model defined in Chapter 3.5.1. We use two- and three-variable universal models for our counter formula benchmarks and universal models with between one and three variables for our random formula benchmarks. For our pattern formula benchmarks, as for the counter and random benchmarks, the number of variables in the universal model used for any particular test was equal to the number of variables in the LTL specification under test.

4.3.4 Symbolic Tools

We compare the explicit tools with three symbolic tools: CadenceSMV, NuSMV, and SAL-SMC. To check whether an LTL formula φ is satisfiable, we model check $\neg\varphi$ against a symbolic universal model. In our benchmark tests, we use the symbolic universal models defined in Chapter 3.5.2, matching the number of variables in the universal model to the number of variables in the LTL formula under test.

All three symbolic model checkers negate the specification, $\neg\varphi$, symbolically compile φ into A_φ, and conjoin A_φ with the given universal model. Then they search for a counterexample in the form of a fair path in the resulting symbolic automaton. If the automaton

---

[13]An interesting alternative to Spin's nested depth-first search algorithm [60] would be to use SPOT's SCC-based search algorithm [187].

is not empty, the symbolic model checker under test returns a computation that satisfies the formula φ. Note that CadenceSMV, NuSMV, and SAL-SMC all act as both a symbolic LTL-to-automaton compiler and a search engine seeking a satisfying counterexample to the input LTL formula. We do not separate these two actions as we do with the explicit LTL-to-automaton tools, all of which produce Promela automata that are checked by Spin.

4.4   The Scalability Challenge

When checking the satisfiability of specifications we need to consider large LTL formulas. Our experiments focus on challenging the tools with scalable formulas. Unfortunately, most explicit tools do not rise to the challenge. In general, the performance of explicit tools degrades substantially as the automata they generate grow beyond 1,000 states. This degradation is manifested in both timeouts (our timeout bound was 4 hours per formula) and errors due to memory management. This should be contrasted with symbolic BDD-based tools, which routinely handle hundreds of thousands and even millions of nodes.

We illustrate this first with run-time results for counter formulas. We display each tool's total run time, which is a combination of the tool's automaton generation time and Spin's model analysis time. We include only data points for which the tools provide correct answers; we know all counter formulas are uniquely satisfiable. As is shown in Figures 4.1 and 4.2,[14] SPOT is the only explicit tool that is somewhat competitive with the symbolic tools. Generally, the explicit tools time out or die before scaling to $n = 10$, when the automata have only a few thousands states; only a few tools passed $n = 8$.

We also found that SAL-SMC does not scale. Figure 4.3 demonstrates that, despite median run times that are comparable with the fastest explicit-state tools, SAL-SMC does not scale past $n = 8$ for any of the counter formulas. No matter how the formula is specified,

---

[14]We recommend viewing all figures online, in color, and magnified.
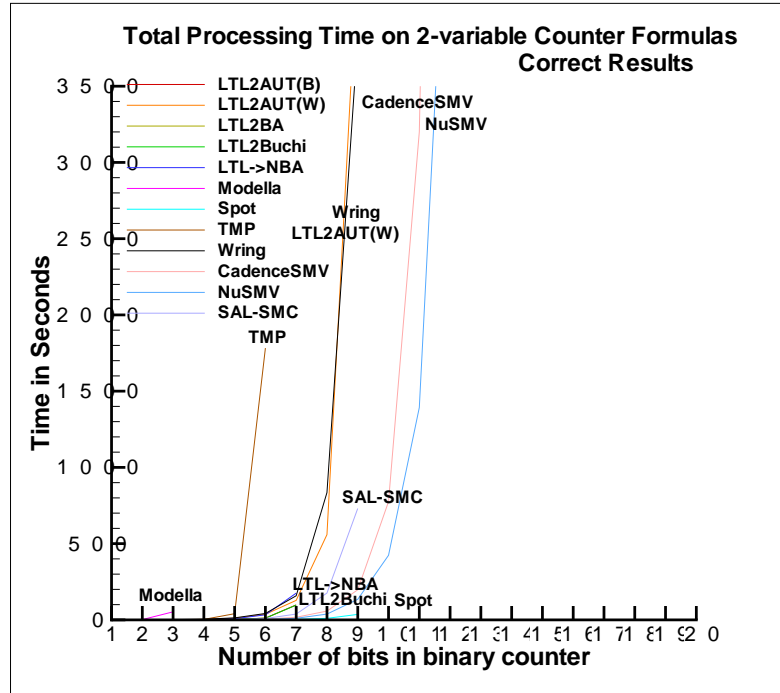
Figure 4.1 : Graph showing the total processing time for 2-variable counter formulas when correct results were obtained, based on the number of bits in the binary counter.
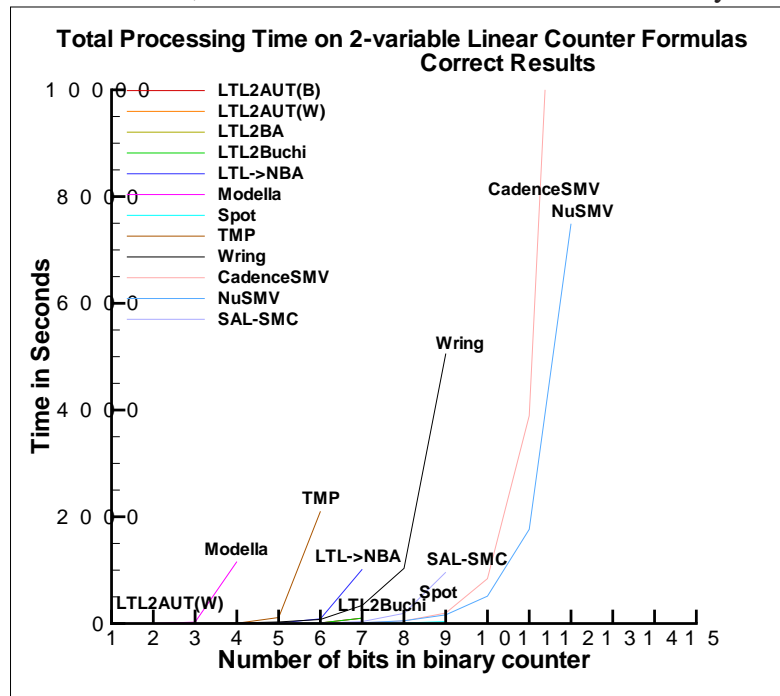
Figure 4.2 : Graph showing the total processing time for 2-variable linear counter formulas when correct results were obtained, based on the number of bits in the binary counter.
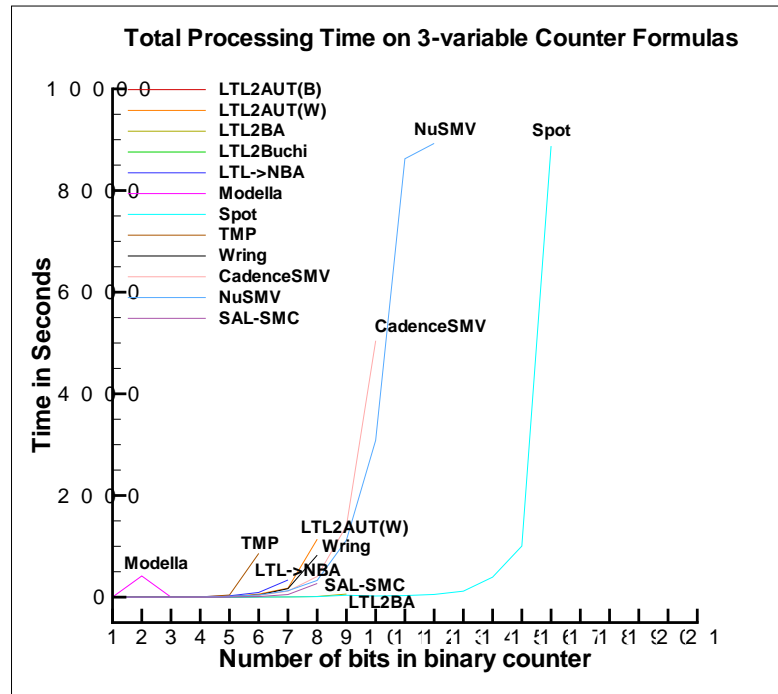
Figure 4.3 : Graph showing the total processing time for 3-variable linear counter formulas based on the number of bits in the binary counter.

SAL-SMC exits with the message "Error: vector too large" when the state space increases from $2^8 \times 8 = 2048$ states at $n = 8$ to $2^9 \times 9 = 4608$ states at $n = 9$. SAL-SMC's behavior on pattern formulas was similar (see Figures 4.6 and 4.11). While SAL-SMC consistently found correct answers, avoided timing out, and always exited gracefully, it does not seem to be an appropriate choice for formulas involving large state spaces. (SAL-SMC has the added inconvenience that it parses LTL formulas differently than all of the other tools described in this chapter: it treats all temporal operators as prefix, instead of infix, operators.)

Figures 4.4 and 4.5 show median automata generation and model analysis times for random formulas. Most tools, with the exception of SPOT and LTL2BA, timeout or die before scaling to formulas of length 60. The difference in performance between SPOT and

LTL2BA, on one hand, and the rest of the explicit tools is quite dramatic. Note that up to length 60, model-analysis time is negligible. SPOT and LTL2BA can routinely handle formulas of up to length 150, while CadenceSMV and NuSMV scale past length 200 with run times of a few seconds.

Figure 4.6 shows performance on the E pattern formulas. Recall that $E(n) = \bigwedge_{i=1}^{n} \Diamond p_i$. The minimally-sized automaton representing $E(n)$ has exactly 2^n states in order to remember which p_i's have been observed. (Basically, we must declare a state for every combination of p_i's seen so far.) However, none of the explicit tools create minimally sized automata. Again, we see all of the explicit tools do not scale beyond $n = 10$, which is minimally 1024 states, in sharp contrast to the symbolic tools.

4.5 Graceless Degradation

Most explicit tools do not behave robustly and die gracelessly. When LTL2Buchi has difficulty processing a formula, it produces over 1,000 lines of `java.lang.StackOverflow Error` exceptions. LTL2BA periodically exits with "Command exited with non-zero status 1" and prints into the Promela file, "ltl2ba: releasing a free block, saw 'end of formula'." Python traceback errors hinder LTL→NBA. Modella suffers from a variety of memory errors including `*** glibc detected *** double free or corruption (out): 0x 55ff4008 ***`. Sometimes Modella causes a segmentation fault and other times Modella dies gracefully, reporting "full memory" before exiting. When used purely as an LTL-to-automaton translator, Spin often runs for thousands of seconds and then exits with non-zero status 1. TMP behaves similarly. Wring often triggers Perl "Use of freed value in iteration" errors. When the translation results in large Promela models, Spin frequently yields segmentation faults during the compilation stage. For example, SPOT translates the formula $E(8)$ to an automaton with 258 states and 6,817 transitions in 0.88 seconds. Spin
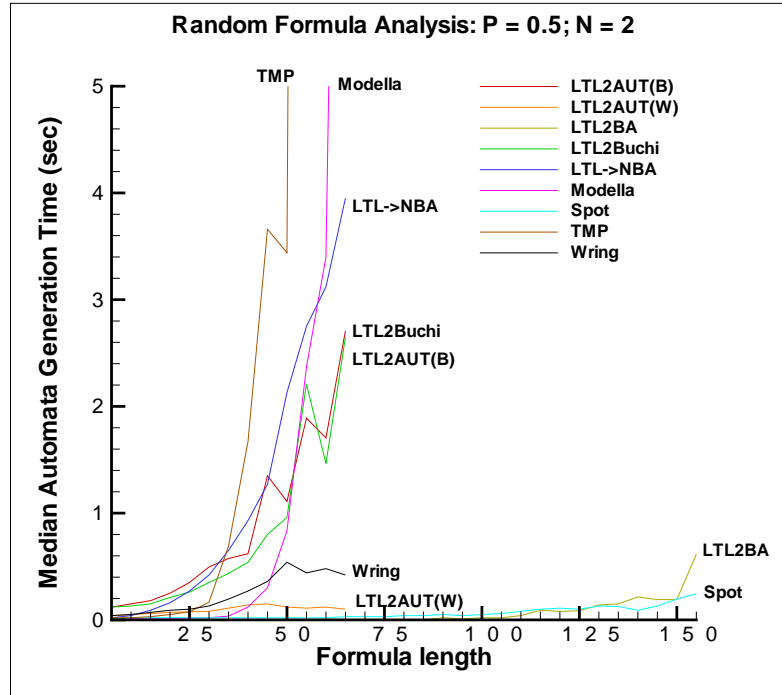
Figure 4.4 : Median automated generation times for random formulas with $P = 0.5$ and $N = 2$ based on formula length.
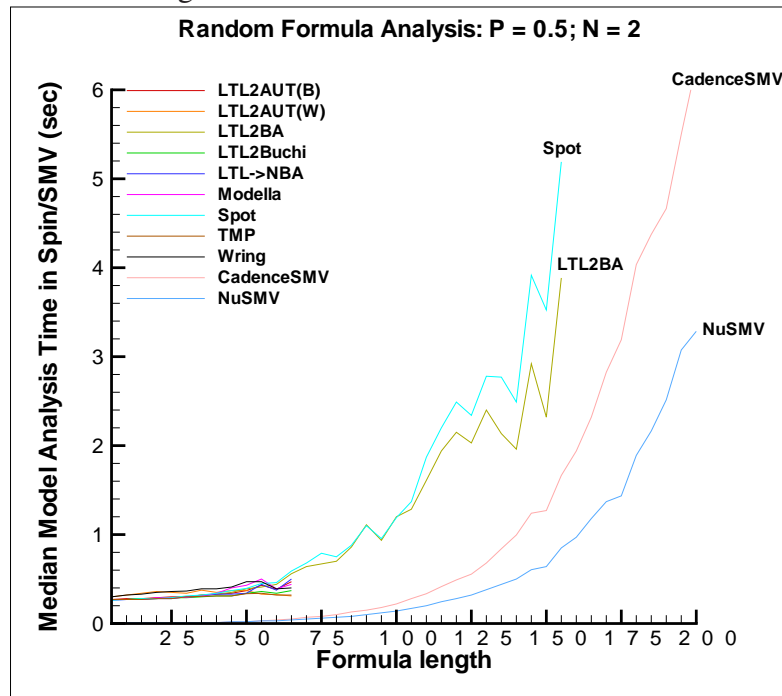
Figure 4.5 : Median model analysis times for random formulas with $P = 0.5$ and $N = 2$ based on formula length.
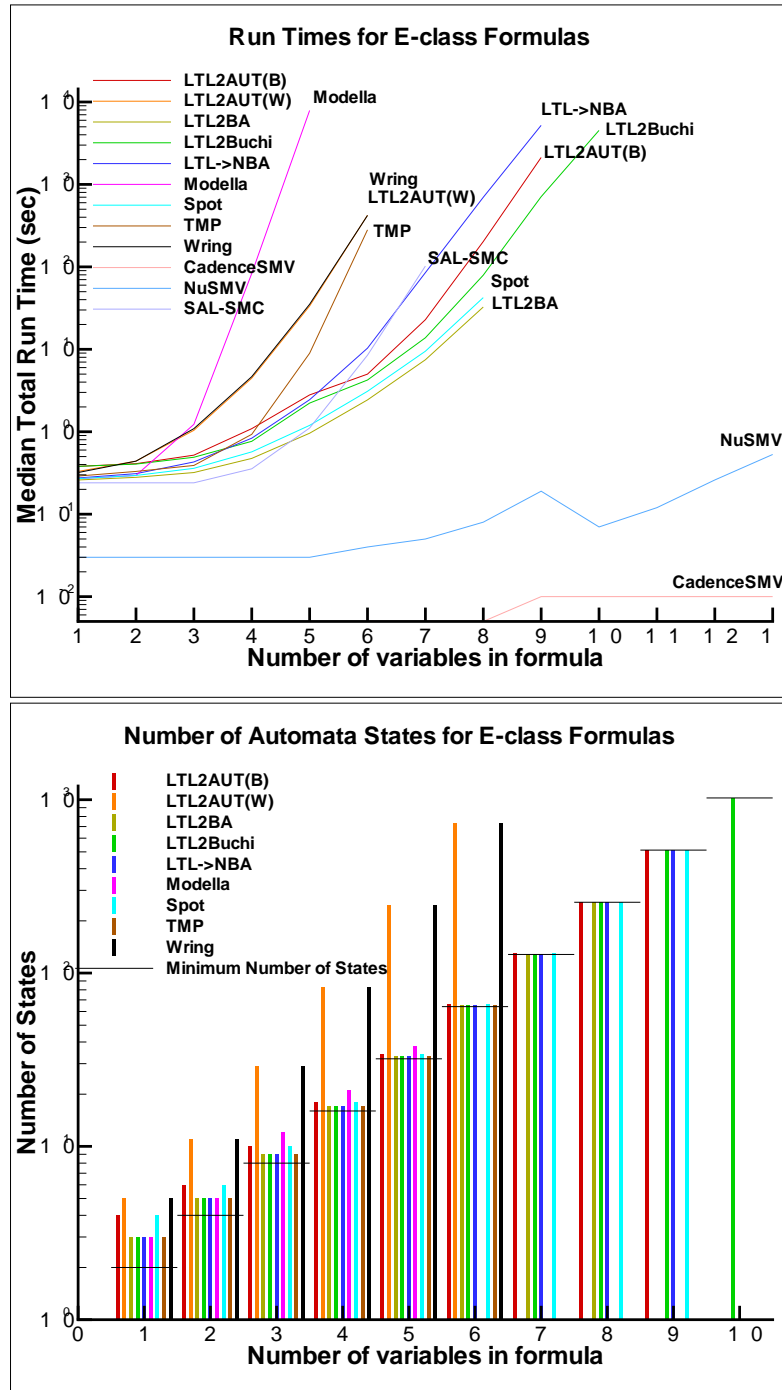
Figure 4.6 : Median total run times and number of automata states for *E* class formulas, based on the number of variables in the formula.

analyzes the resulting Promela model in 41.75 seconds. SPOT translates the $E(9)$ formula to an automaton with 514 states and 20,195 transitions in 2.88 seconds, but Spin segmentation faults when trying to compile this model. SPOT and the three symbolic tools are the only tools that consistently degrade gracefully; they either timeout or terminate with a succinct, descriptive message.

A more serious problem is that of incorrect results, i.e., reporting "satisfiable" for an unsatisfiable formula or vice versa. The problem is particularly acute when the returned automaton A_φ is empty (has no states). On one hand, an empty automaton accepts the empty language. On the other hand, Spin conjoins the Promela model for the `never` claim with the model under verification, so an empty automaton, when conjoined with a universal model, actually acts as a universal model. The tools are not consistent in their handling of empty automata. Some, such as LTL2Buchi and SPOT return an explicit indication of an empty automaton, while Modella and TMP just return an empty Promela model. We have taken an empty automaton to mean "unsatisfiable." In Figure 4.7 we show an analysis of correctness for random formulas. Here we counted "correct" as any verdict, either "satisfiable" or "unsatisfiable," that matched the verdict found by both CadenceSMV and NuSMV for the same formula; we found that CadenceSMV and NuSMV always agree with each other. We excluded data for any formulas that timed out or triggered error messages. All of the explicit tools showed degraded correctness as the formulas scale in size, in sharp contrast to the symbolic tools, all of which reported correct results for every test.[15]

---

[15]Note that this is no longer the case today as shortly following our initial study, the subtle bug we uncovered in the SPOT translator was corrected by the tool author, who also integrated our benchmarks into SPOT's test suite. Today, SPOT, like the symbolic tools, always produces correct automata.
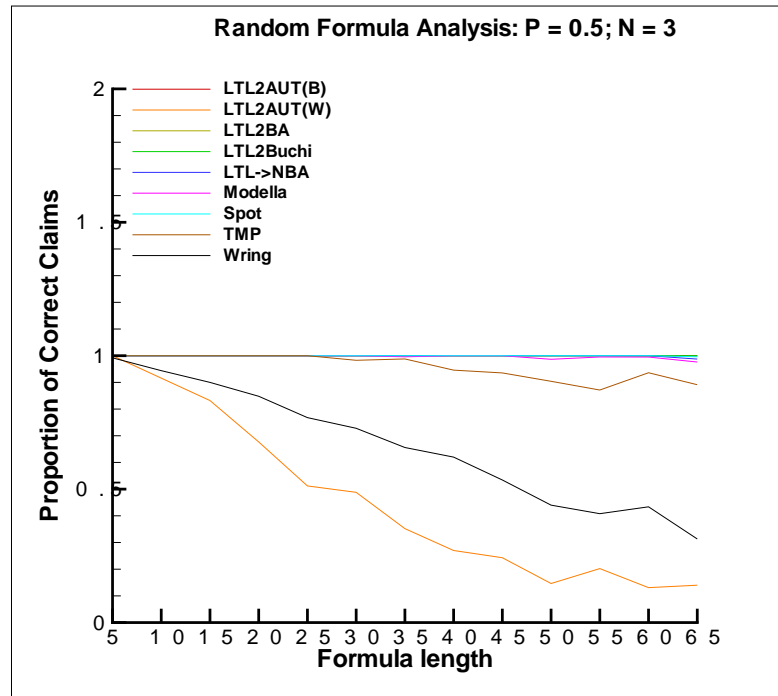
Figure 4.7 : Graph showing the degredation of proportion of correct claims for random formulas where $P = 0.5$ and $N = 3$ based on the length of the random formula.

4.6   Relation of Size to Efficiency

The focus of almost all LTL translation papers, starting with [134], has been on minimizing automaton size. It has already been noted that automata minimization may not result in model checking performance improvement [186] and specific attention has been given to minimizing the size of the product with the model [139, 64]. Our results show that size, in terms of both number of automaton states and transitions, is not a reliable indicator of satisfiability checking run time. Intuitively, the smaller the automaton, the easier it is to check for nonemptiness. This simplistic view, however, ignores the effort required to minimize the automaton. It is often the case that tools spend more time constructing the formula automaton than constructing and analyzing the product automaton. As an example, consider the performance of the tools on counter formulas. We see in Figures 4.1 and 4.2

dramatic differences in the performance of the tools on such formulas. In contrast, we see in Figures 4.8 and 4.9 that the tools do not differ significantly in terms of the sizes of the generated automata. (For reference, we have marked on these graphs the minimum automaton size for an n-bit binary counter, which is $(2^n) * n + 1$ states. There are 2^n numbers in the series of n bits each plus one additional initial state, which is needed to assure the automaton does not accept the empty string.) Similarly, Figure 4.6 shows little correlation between automaton size and run time for E pattern formulas.

Consider also the performance of the tools on random formulas. In Figure 4.10 we see the performance in terms of the sizes of the generated automata. Performance in terms of run time is plotted in Figure 4.12, where each tool was run until it timed out or reported an error for more than 10% of the sampled formulas. SPOT and LTL2BA consistently have the best performance in terms of total run time, but they are average performers in terms of automaton size. LTL2Buchi consistently produces significantly more compact automata, in terms of both states and transitions. It also incurs lower Spin model analysis times than SPOT and LTL2BA. Yet LTL2Buchi spends so much time generating the automata that it does not scale nearly as well as SPOT and LTL2BA.

4.7 Symbolic Approaches Outperform Explicit Approaches

Across the various classes of formulas, the symbolic tools outperformed the explicit tools, demonstrating faster performance and increased scalability. (We measured only combined automata-generation and model-analysis time for the symbolic tools. The translation to automata is symbolic and is very fast; it is linear in the size of the formula [68].) We see this dominance with respect to counter formulas in Figures 4.1 and 4.2, for random formulas in Figures 4.4, 4.5, and 4.12, and for E pattern formulas in Figure 4.6. For U pattern formulas, no explicit tools could handle $n = 10$, while CadenceSMV and NuSMV
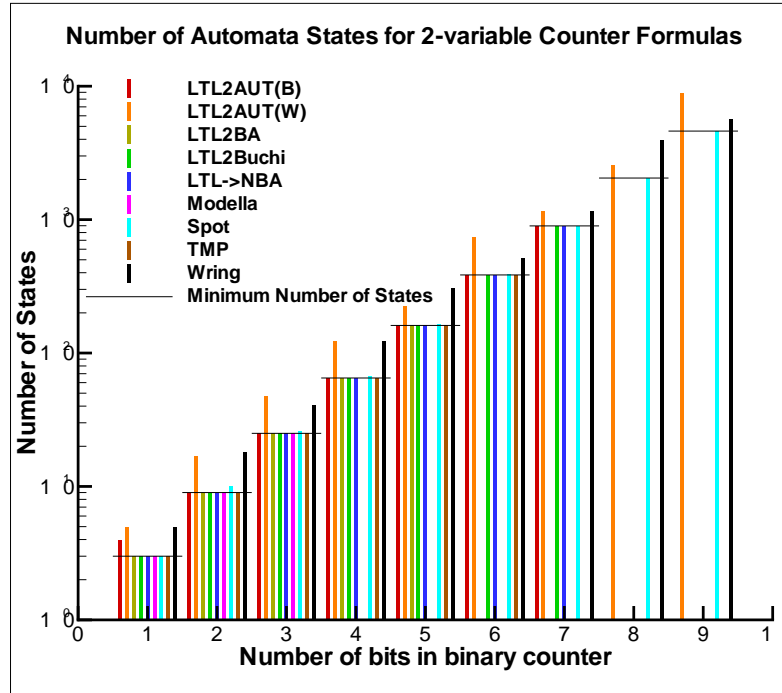
Figure 4.8 : Graph of the number of states in the automata representing 2-variable counter formulas, based on the number of bits in the binary counter.
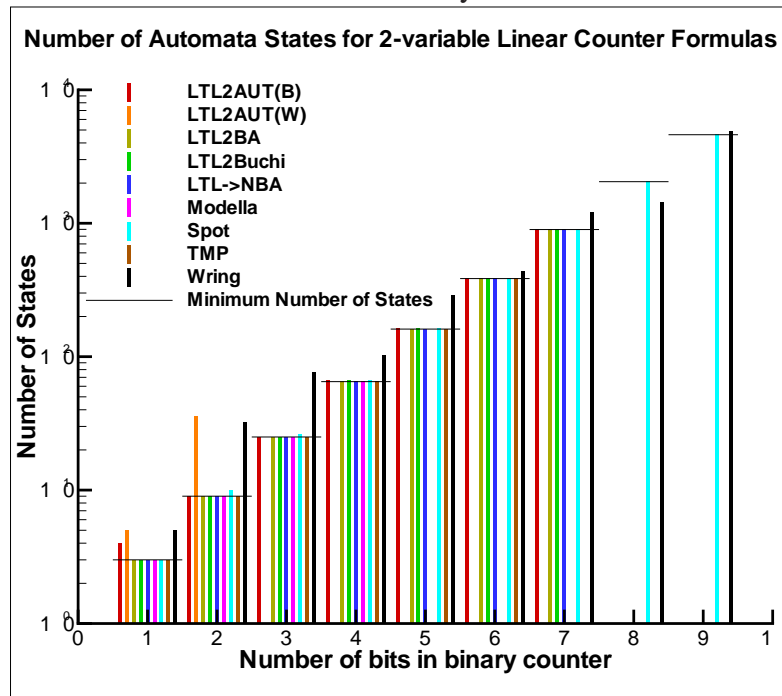
Figure 4.9 : Graph of the number of states in the automata representing 2-variable linear counter formulas, based on the number of bits in the binary counter.
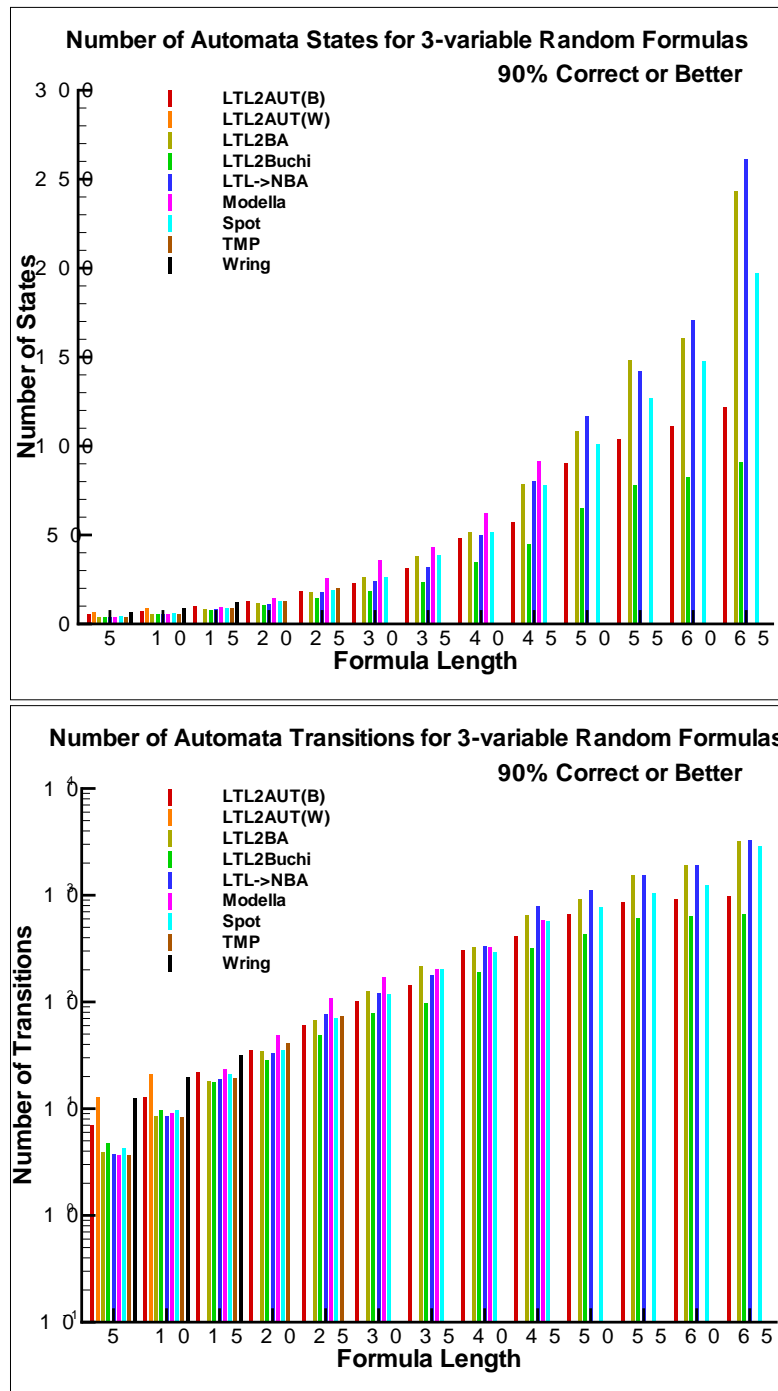
Figure 4.10 : Graphs showing the number of states and transitions in the automata representing 3-variable random counter variables, based on the length of the formula when correctness was 90% or better.

scale up to $n = 20$; see Figure 4.11. Recall that $U(n) = (\ldots(p_1 \; \mathcal{U} \; p_2) \; \mathcal{U} \; \ldots) \; \mathcal{U} \; p_n$, so while there is not a clear, canonical automaton for each U pattern formula, it is clear that the automaton size is exponential.

The only exception to the dominance of the symbolic tools occurs with 3-variable linear counter formulas, where SPOT outperforms all symbolic tools. We ran the tools on many thousands of formulas and did not find a single case in which any symbolic tool yielded an incorrect answer yet every explicit tool gave at least one incorrect answer during our tests.

The dominance of the symbolic approach is consistent with the findings in [197, 198], which reported on the superiority of a symbolic approach with respect to an explicit approach for satisfiability checking for the modal logic K. In contrast, [199] compared explicit and symbolic translations of LTL to automata in the context of symbolic model checking and found that explicit translation performs better in that context. Consequently, [199] advocates a *hybrid* approach, combining symbolic systems and explicit automata. Note, however, that not only is the context in [199] different than here (model checking rather than satisfiability checking), but also the formulas studied there are generally small and translation time is negligible, in sharp contrast to the study we present here. We return to the topic of model checking in the concluding discussion.

Figures 4.4, 4.5, and 4.12 reveal why the explicit tools generally perform poorly. We see in these figures that for most explicit tools automata-generation times by far dominate model-analysis times, which calls into question the focus in the literature on minimizing automaton size. Among the explicit tools, only SPOT and LTL2BA seem to have been designed with execution speed in mind. Note that, other than Modella, SPOT and LTL2BA are the only tools implemented in C/C++.
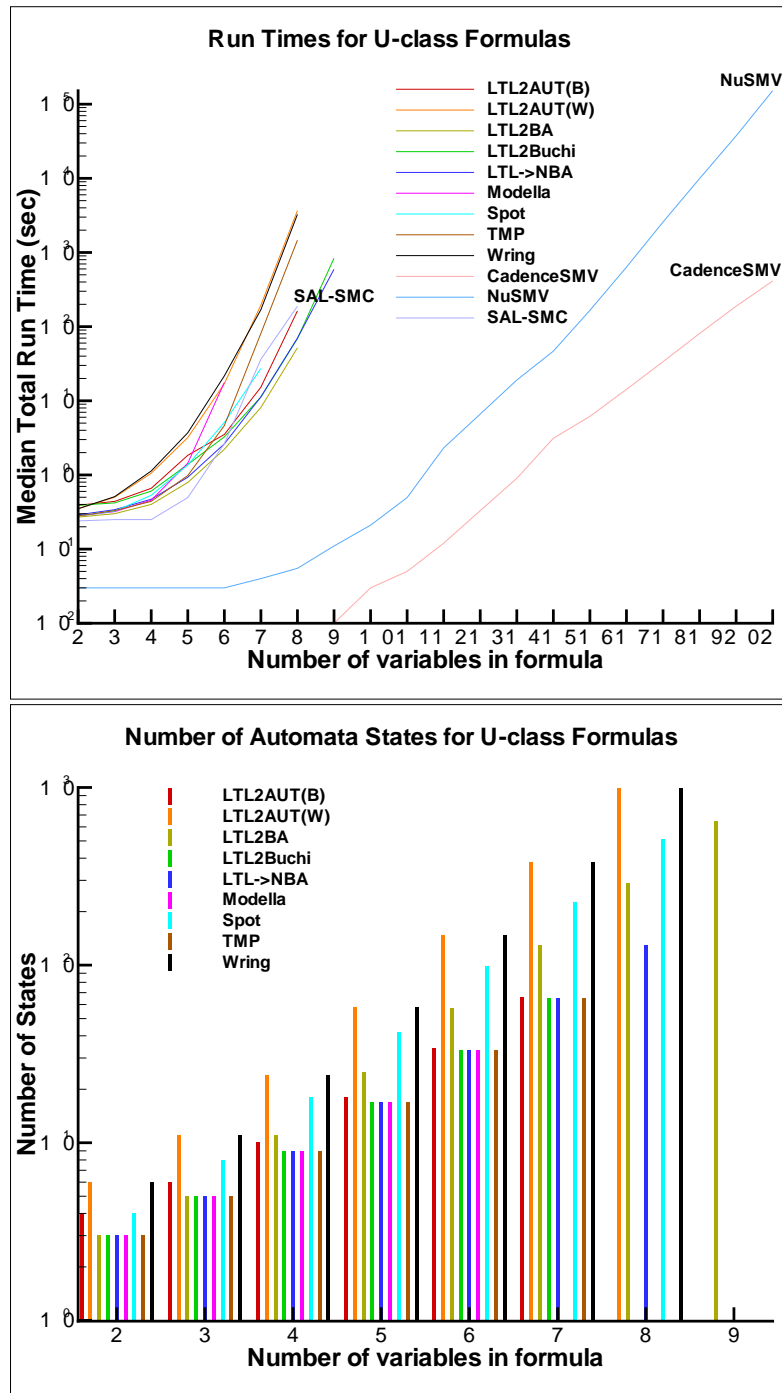
Figure 4.11 : Graphs showing the median total run time and the number of states in the automata representing *U*-class, based on the number of variables in the formulas.
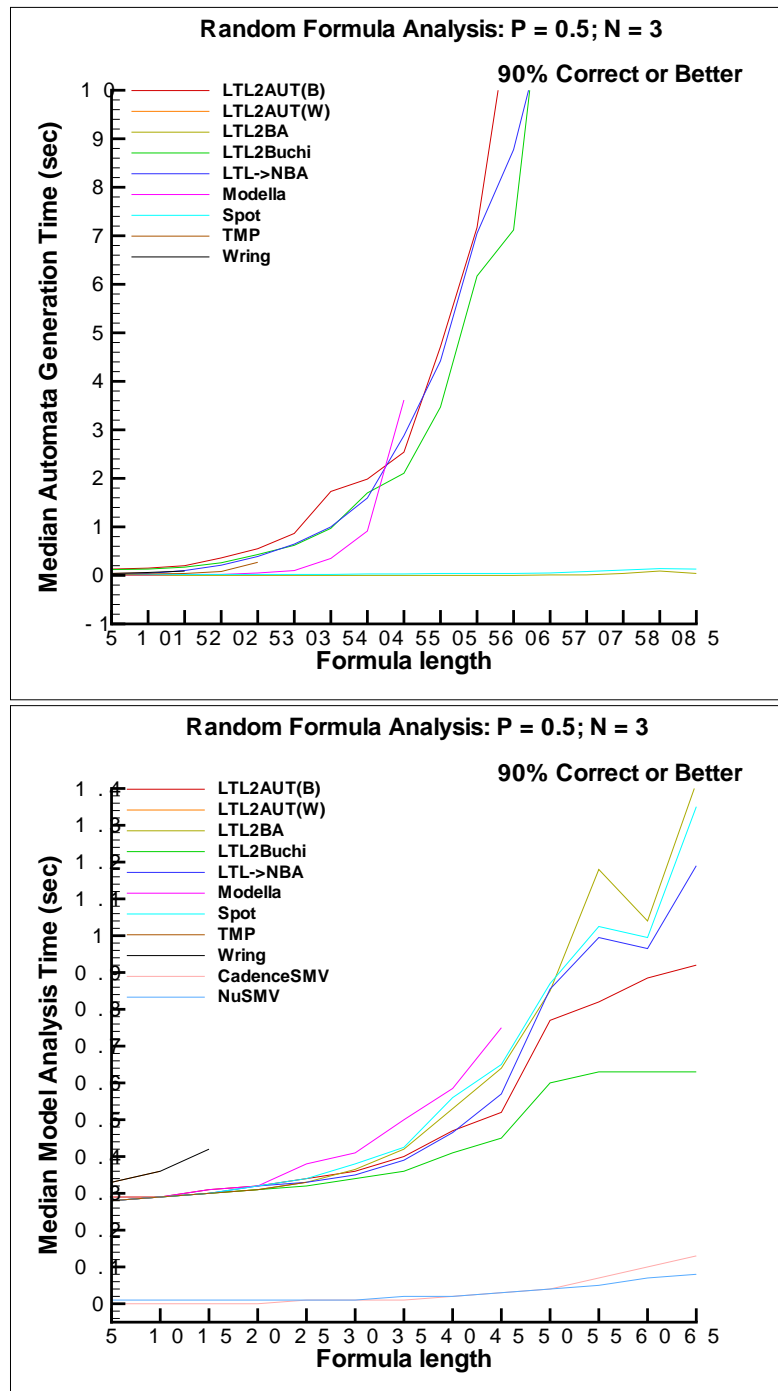
Figure 4.12 : Graphs showing the median automata generation times and the median model analysis times for random formulas with $P = 0.5$ and $N = 3$, based on the length of the formula when 90% or better are correct.

4.8 Discussion

Too little attention has been given in the formal-verification literature to the issue of debugging specifications. We argued here for the adoption of a basic sanity check: satisfiability checking for each specification, its compliment, and the conjunction of all specifications. We showed that LTL satisfiability checking can be done via a reduction to model checking universal models and benchmarked a large array of tools with respect to satisfiability checking of scalable LTL formulas.

We found that the existing literature on LTL to automata translation provides little information on actual tool performance. We showed that most explicit LTL translation tools, with the exception of SPOT, are research prototypes, which cannot be considered industrial-quality tools. The focus in the literature has been on minimizing automaton size, rather than evaluating overall performance. Focusing on overall performance reveals a large difference between LTL translation tools. In particular, we showed that symbolic tools have a clear edge over explicit tools with respect to LTL satisfiability checking.

While the focus of our study was on LTL satisfiability checking, there are a couple of conclusions that apply to model checking in general. First, LTL translation tools need to be fast and robust. In our judgment, this rules out implementations in languages such as Perl or Python and favors C or C++ implementations. Furthermore, attention needs to be given to graceful degradation. In our experience, tool errors are invariably the result of graceless degradation due to poor memory management. Second, tool developers should focus on overall performance instead of output size. It has already been noted that automata minimization may not result in model checking performance improvement [186] and specific attention has been given to minimizing the size of the product with the model [139]. Still, no previous study of LTL translation has focused on model checking performance, leaving a glaring gap in our understanding of LTL model checking.

Chapter 5

A Multi-Encoding Approach for LTL Symbolic Satisfiability Checking

5.1 Introduction

In *property-based design* formal properties, written in temporal logics such as LTL [42], are written early in the system-design process and communicated across all design phases to increase the efficiency, consistency, and quality of the system under development [11, 12]. Property-based design and other design-for-verification techniques capture design intent precisely, and use formal logic properties both to guide the design process and to integrate verification into the design process [10]. The shift to specifying desired system behavior in terms of formal logic properties risks introducing specification errors in this very initial phase of system design, raising the need for *property assurance* [12, 47].

The need for checking for errors in formal LTL properties expressing desired system behavior first arose in the context of model checking, where *vacuity checking* aims at reducing the likelihood that a property that is satisfied by the model under verification is an erroneous property [48, 49]. Property assurance is more challenging at the initial phases of property-based design, before a model of the implementation has been specified. *Inherent vacuity checking* is a set of sanity checks that can be applied to a set of temporal properties, even before a model of the system has been developed, but many possible errors cannot be detected by inherent vacuity checking [50].

A stronger sanity check for a set of temporal properties is LTL *realizability checking*, in which we test whether there is an open system that satisfies all of the properties in the

set [51], but such a test is very expensive computationally. In LTL *satisfiability checking*, we test whether there is a closed system that satisfies all of the properties in the set. The satisfiability test is weaker than the realizability test, but its complexity is lower; it has the same complexity as LTL model checking [46]. In fact, LTL satisfiability checking can be implemented via LTL model checking; see Chapter 2.3.

Indeed, the need for LTL satisfiability checking is widely recognized [54, 55, 56, 57, 58]. Foremost, it serves to ensure that the behavioral description of a system is internally consistent and neither over- or under-constrained. If an LTL property is either *valid*, or *unsatisfiable* this must be due to an error. Consider, for example, the specification *always* ($b_1 \rightarrow$ *eventually* b_2), where b_1 and b_2 are propositional formulas. If b_2 is a tautology, then this property is valid. If b_2 is a contradiction, then this property is unsatisfiable. Furthermore, the collective set of properties describing a system must be satisfiable, to avoid contradictions between different requirements. Satisfiability checking is particularly important when the set of properties describing the design intent continues to evolve, as properties are added and refined, and have to be checked repeatedly. Because of the need to consider large sets of properties, it is critical that the satisfiability test be *scalable*, and able to handle complex temporal properties. This is challenging, as LTL satisfiability is known to be PSPACE-complete [46].

As pointed out in Chapter 4 and [56], satisfiability checking can be performed via model checking. In Chapter 4 and [56] we explored the effectiveness of model checkers as LTL satisfiability checkers. We compared there the performance of explicit-state and symbolic model checkers. Both use the automata-theoretic approach [39] but in a different way. Explicit-state model checkers translate LTL formulas to Büchi automata explicitly and then use an explicit graph-search algorithm [41], as described in Chapter 2.8. For satisfiability checking, the construction of the automaton is the more demanding task. Symbolic model checkers construct symbolic encodings of automata and then use a symbolic nonemptiness

test. The symbolic construction of the automaton is easy, but the nonemptiness test is computationally demanding. The extensive set of experiments described in Chapter 4 and [56] showed that the symbolic approach to LTL satisfiability is significantly superior to the explicit-state approach in terms of scalability.

In the context of explicit-state model checking, there has been extensive research on optimized construction of automata from LTL formulas [130, 135, 138, 137, 134, 136, 139, 63], where a typical goal is to minimize the size of constructed automata [40]. Optimizing the construction of symbolic automata is more difficult, as the size of the symbolic representation does not correspond directly to its optimality. An initial symbolic encoding of automata was proposed in [67], but the optimized encoding we call *CGH*, proposed by Clarke, Grumberg, and Hamaguchi [68], has become the de facto standard encoding. The CGH encoding is used by model checkers such as CadenceSMV and NuSMV, and has been extended to symbolic encodings of industrial specification languages [204]. Surprisingly, there has been little follow-up research on this topic.

In this chapter, we propose novel symbolic LTL-to-automaton translations and utilize them in a new multi-encoding approach to achieve significant, sometimes exponential, improvement over the current standard encoding for LTL satisfiability checking. First we introduce and prove the correctness of a novel encoding of symbolic automata inspired by optimized constructions of explicit automata [130, 138]. While the CGH encoding uses *Generalized Büchi Automata* (GBA), our new encoding is based on *Transition-based Generalized Büchi Automata* (TGBA). Second, inspired by work on symbolic satisfiability checking for modal logic [197], we introduce here a novel *sloppy* encoding of symbolic automata, as opposed to the *fussy* encoding used in CGH. Sloppy encoding uses looser constraints, which sometimes results in smaller BDDs. The sloppy approach can be applied both to GBA-based and TGBA-based encodings, provided that one uses negation-normal form (NNF), [136], rather than the Boolean normal form (BNF) used in CGH. Finally, we

introduce several new variable-ordering schemes, based on tree decompositions of the LTL parse tree corresponding to a given formula, and inspired by observations that relate tree decompositions to BDD variable ordering [205]. The combination of GBA/TGBA, fussy/sloppy, BNF/NNF, and different variable orders yields a space of 30 possible configurations of symbolic automata encodings. (Not all combinations yield viable configurations.)

Since the value of novel encoding techniques lies in increased *scalability*, we evaluate our novel encodings in the context of LTL satisfiability checking, utilizing a comprehensive and challenging collection of widely-used benchmark formulas [56, 57, 206, 58]; see also Chapter 3. For each formula, we perform satisfiability checking using all 30 encodings. (We use CadenceSMV as our experimental platform.) Our results demonstrate conclusively that no encoding performs best across our large benchmark suite. Furthermore, no single approach–GBA vs. TGBA, fussy vs. sloppy, BNF vs. NNF, or any one variable order, is dominant. This is consistent with the observation made by others [207, 40] that in the context of symbolic techniques one typically does not find a "winning" algorithmic configuration. In response, we developed a multi-encoding tool, PANDA, which runs several encodings in parallel, terminating when the first process returns. We call our tool PANDA for "Portfolio Approach to Navigate the Design of Automata." Our experiments demonstrate conclusively that the multi-encoding approach using the novel encodings invented in this chapter achieves substantial improvement over CGH, the current standard encoding; in fact PANDA significantly bested the native LTL model checker built into CadenceSMV.

The structure of this chapter is as follows. We review the CGH encoding [68] in Section 5.2. Next, in Section 5.3, we describe our novel symbolic TGBA encoding. We introduce our novel sloppy encoding and our new methods for choosing BDD variable orders and discuss our space of symbolic encoding techniques in Section 5.4. After setting up our scalability experiment in Section 5.5, we present our test results in Section 5.6, followed by a discussion in Section 5.7. Though our construction can be used with different symbolic

model checking tools, in this chapter, we follow the convention of [68] and give examples of all constructions using the SMV syntax.

5.2 Preliminaries

We assume familiarity with LTL [87]; recall that Definition 2 defines LTL semantics. We also assume familiarity with Generalized Büchi Automata; recall that they are defined in Chapter 2.4.

We use two normal forms:

Definition 9

Boolean Normal Form (BNF) *rewrites the input formula to use only* $\neg$, $\vee$, $\mathcal{X}$, $\mathcal{U}$, *and* $\mathcal{F}$. *In other words, we replace* $\wedge$, $\rightarrow$, $\mathcal{R}$, *and* $\mathcal{G}$ *with their equivalents:*

$$g_1 \wedge g_2 \equiv \neg(\neg g_1 \vee \neg g_2) \qquad\qquad g_1 \,\mathcal{R}\, g_2 \equiv \neg(\neg g_1 \,\mathcal{U}\, \neg g_2)$$

$$g_1 \rightarrow g_2 \equiv \neg g_1 \vee g \qquad\qquad \mathcal{G}g_1 \equiv \neg\mathcal{F}\neg g_1$$

Definition 10

Negation Normal Form (NNF) *pushes negation inwards until only atomic propositions are negated, using the following rules:*

$$\neg(\mathcal{X}g) \equiv \mathcal{X}(\neg g)$$

$$\neg\neg g \equiv g \qquad\qquad \neg(g_1 \mathcal{U} g_2) \equiv (\neg g_1 \mathcal{R} \neg g_2)$$

$$\neg(g_1 \wedge g_2) \equiv (\neg g_1) \vee (\neg g_2) \qquad\qquad \neg(g_1 \mathcal{R} g_2) \equiv (\neg g_1 \mathcal{U} \neg g_2)$$

$$\neg(g_1 \vee g_2) \equiv (\neg g_1) \wedge (\neg g_2) \qquad\qquad \neg(\mathcal{G}g) \equiv \mathcal{F}(\neg g)$$

$$(g_1 \rightarrow g_2) \equiv (\neg g_1) \vee g_2 \qquad\qquad \neg(\mathcal{F}g) \equiv \mathcal{G}(\neg g)$$

Recall from Chapter 2.2.1 that a trace satisfying LTL formula f is an infinite run over the alphabet $\Sigma = 2^{Prop}$, where *Prop* is the underlying set of atomic propositions. We denote by *models*(f) the set of traces satisfying f. The next theorem (a restatement of Theorem 2.1) relates the expressive power of LTL to that of Büchi automata.

Theorem 5.1

[44] *Given an LTL formula f, we can construct a generalized Büchi automaton $A_f = \langle Q, \Sigma, \delta, Q_0, F \rangle$ such that $|Q|$ is in $2^{O(|f|)}$, $\Sigma = 2^{Prop}$, and $\mathcal{L}_\omega(A_f)$ is exactly models(f).*

This theorem, proved in Chapter 2.4, reduces LTL satisfiability checking to automata-theoretic nonemptiness checking, as f is satisfiable iff *models*(f) $\neq \emptyset$ iff $\mathcal{L}_\omega(A_f) \neq \emptyset$.

Recall from Chapter 2.3 that LTL satisfiability checking relates to LTL model checking as follows. We use a *universal model M* that generates all traces over *Prop* such that $\mathcal{L}_\omega(M) = (2^{Prop})^\omega$. The code for this model appears in [56] and Chapter 3.5.2. We now have that M does *not* satisfy $\neg f$ iff f is satisfiable. We use a symbolic model checker to check the formula $\neg f$ against M; f is satisfiable precisely when the model checker finds a counterexample.

CGH encoding   In this chapter we focus on LTL to symbolic Büchi automata compilation. We recap the CGH encoding [68], which assumes that the formula f is in BNF, and then forms a symbolic GBA. We first define the *CGH-closure* of an LTL formula f as the set of all subformulas of f (including f itself), where we also add the formula $X(g \mathcal{U} h)$ for each subformula of the form $g \mathcal{U} h$. The X-formulas in the CGH-closure of f are called *elementary* formulas.

We declare a Boolean SMV variable EL_{Xg} for each elementary formula Xg in the CGH-closure of f. Also, each atomic proposition in f is declared as a Boolean SMV variable. We define an auxiliary variable S_h for every formula h in the CGH-closure of f. (Auxiliary variables are substituted away by SMV and do not require allocated BDD variables.) The

characteristic function for an auxiliary variable S_h is defined as follows:

$$S_h = p \quad \text{if} \quad p \in AP$$

$$S_h = !S_g \quad \text{if} \quad h = \neg g$$

$$S_h = EL_h \quad \text{if} \quad h \text{ is a formula } \mathcal{X}g$$

$$S_h = S_{g1}|S_{g2} \quad \text{if} \quad h = g_1 \vee g_2$$

$$S_h = S_{g2}|(S_{g1}\&S_{\mathcal{X}_{(g1 \; \mathcal{U} \; g2)}}) \quad \text{if} \quad h = g_1 \; \mathcal{U} \; g_2$$

We now generate the SMV model M_f:

```
MODULE main

VAR
  /*declare a Boolean variable for each atomic prop in f */
  a: boolean;

  /*declare a Boolean variable for
    every formula Xg in the CGH-closure*/
  EL_Xg: boolean;

DEFINE  /*auxiliary vars according to characteristic function */
  S_h := ...

TRANS /*for every formula Xg in the CGH-closure,
        add a transition constraint*/
  ( S_X_g1 = next(S_g1) ) &
           ⋮
  ( S_X_gn = next(S_gn) )

FAIRNESS  !S_gUh | S_h /*for each subformula gUh */

FAIRNESS  TRUE /*or a generic fairness condition otherwise*/

SPEC      !(S_f & EG true) /*end with a SPEC statement*/
```

The traces of M_f correspond to the accepting runs of A_f, starting from arbitrary states. Thus, satisfiability of f corresponds to nonemptiness of M_f, starting from an initial state. We can model check such nonemptiness with `SPEC !(S_f & EG true)`. A counterexample is an infinite trace starting at a state where S_f holds. Thus, the model checker returns a counterexample that is a trace satisfying f.

Remark 5.1

While the syntax we use is shared by CadenceSMV and NuSMV, the precise semantics of CTL model checking in these model checkers is not fully documented and there are some subtle but significant differences between the two tools. Therefore, we use CadenceSMV semantics here and describe these subtleties below.

5.2.1 CadenceSMV and NuSMV Semantic Subtleties

We encountered one large, and several more subtle but still impactful differences between the implementations of CadenceSMV and NuSMV when testing our novel encodings. Most significantly, TGBA-encoded symbolic automata cannot be checked using NuSMV because variable definitions in terms of `DEFINE` statements may only assign simple expressions composed of state variables [181]. Therefore, NuSMV cannot parse the `next()` operators in our `DEFINE` section. NuSMV does not offer alternative ways to define variables (e.g. `ASSIGN`-statements) that allow our TGBA construction.[1] (While NuSMV `ASSIGN`-statements do allow `next()` operators, they must occur alone on the left-hand-side of the assignment, which still excludes our TGBA construction.) In CadenceSMV, `next()` statements may not be nested or present in `INIT`, `FAIRNESS`, or `SPEC` statements. Our solution is to use `EL`-variables in `INIT` and `SPEC` statements and use Promise variables in `FAIRNESS` statements.

---

[1] Thanks to Viktor Schuppan and the members of the NuSMV team for allowing our construction in future versions of NuSMV.

Though NuSMV cannot parse our TGBA-encoded automata, for all of the encodings we could check with both SMV variants, we saw similar, significant improvements when running PANDA with that model checker as a back-end, versus running the model checker alone. As expected, we found our automata with NuSMV as a back end produced different timing results than the same automata with CadenceSMV as a back end, just as running each of the tools alone produced different timing results, though neither SMV was always faster. Both tools agreed on the satisfiability of a given LTL formula 100% of the time. Also, the results (either SAT or UNSAT) returned by CadenceSMV and NuSMV always agreed with the results of running all 30 encodings of the same formula.

However, in order to compare our novel encodings across both SMV back-ends, we had to account for several non-intuitive subtle semantic differences between CadenceSMV and NuSMV. While both basically use the CGH encoding, the precise semantics of CTL model checking in CadenceSMV and NuSMV are not explicitly documented and the subtle, often unexpected, differences between the implementations of the two tools complicates the problem of creating alternative encodings. The following rules are necessary to work around the several subtleties that arise when checking nonemptiness of fair symbolic Büchi automata.

There must be at least one initial state. For both NuSMV and CadenceSMV, there is an implicit universal quantifier over all initial states. If there are no initial states, then the formula is automatically "true." Declaring an initial state is not enough to satisfy this condition. For example, `INIT (a&(!a))` specifies that there is no initial state. Similar semantic subtleties have impacted related work with SMV, such as model checking temporal logic meta-property specifications for abstract state machines [208], where this unexpected quantification over initial states means that $M \models EF(\varphi) \not\equiv M \not\models AG(\neg\varphi)$.

Symbolic automata must always have a FAIR statement, even if it is "FAIR true." CadenceSMV considers terminal traces to be fair when there are no fairness constraints. A

fair trace is defined to be a maximal trace on which all fairness constraints are true infinitely often, including terminal traces in models without fairness constraints. The semantics with a fairness constraint is the "infinite traces" semantics where states without infinite traces are discarded. Therefore, we must have at least one fairness constraint to prevent the possibility of a model with one initial state and no legal transitions from model checking as "false." Rather than the classical algorithm of implicitly universally quantifying over all initial states, NuSMV restricts itself to all *fair* initial states. If there are no fair initial states, the formula is automatically "true."

The SPEC should be (! ($\varphi \wedge$ EG true)). Both CadenceSMV and NuSMV consider a CTL formula φ to hold in a model M if φ holds in *all* initial states of M. If M has no initial states, then every φ holds in M. Using `SPEC (!($\varphi$ ∧ EG true))`, if the model is not empty, the counterexample returned is a trace of the model.

INIT φ SPEC !(EG true) is not equivalent to SPEC (!($\varphi \wedge$ EG true)). For example, if φ is simply $false$ and we check `SPEC (!($false$ ∧ EG true))`, then the check will pass (i.e. there will be no counterexample indicating satisfiability), since no trace satisfies $false$. If, however, we state `INIT` $false$ and check `SPEC !(EG true)`, then the check will fail and a counterexample will be returned, which is clearly not what we intended. Similarly, checking for a finite counterexample using `SPEC !($S_f$)` may produce spurious results.

5.3 Encoding Symbolic Transition-based Generalized Büchi Automata

We now introduce a novel symbolic encoding, referred to as TGBA, inspired by the explicit-state transition-based generalized Büchi automata of [138]. Such automata are used by SPOT [140], which was shown experimentally [56] to be the best explicit LTL translator for satisfiability checking.

Definition 11

*A **Transition-based Generalized Büchi Automaton** (TGBA) is a quintuple $(Q, \Sigma, \delta, Q_0, F)$, where:*

- *$\delta \subseteq Q \times \Sigma \times Q$ is a transition relation.*
- *Q is a finite set of states.*
- *$Q_0 \subseteq Q$ is a set of initial states.*
- *Σ is a finite alphabet.*
- *$F \subseteq 2^\delta$ is a set of accepting transitions.*

A run of a TGBA over an infinite trace $\pi = \pi_0, \pi_1, \pi_2, \ldots \in \Sigma$ is a sequence $\langle q_0, \pi_0, q_1 \rangle$, $\langle q_1, \pi_1, q_2 \rangle, \langle q_2, \pi_2, q_3 \rangle, \ldots$ of transitions in δ such that $q_0 \in Q_0$ and $q_i \in Q$. The automaton accepts π if it has a run over π that traverses some transition from each set in F infinitely often.

The next theorem relates the expressive power of LTL to that of TGBAs.

Theorem 5.2

[130, 138] *Given an LTL formula f, we can construct a TGBA $A_f = \langle Q, \Sigma, \delta, Q_0, F \rangle$ such that $|Q|$ is in $2^{O(|f|)}$, $\Sigma = 2^{Prop}$, and $\mathcal{L}_\omega(A_f)$ is exactly models(f).*

Expressing acceptance conditions in terms of transitions rather than states enables a significant reduction in the size of the automata corresponding to LTL formulas [130, 138].

Our new encoding of symbolic automata, based on TGBAs, requires that the input formula f is in NNF; encoding formulas in BNF or other forms may produce spurious results. (This is due to the way that the satisfaction of $\mathcal{U}$-formulas is handled by means of promise variables; see below.) As in CGH, we first define the *closure* of an LTL formula f. In the case of TGBAs, however, we simply define the closure to be the set of all subformulas of f (including f itself). Note that, unlike in the CGH encoding, $\mathcal{U}$- and $\mathcal{F}$- formulas do not require the introduction of new $\mathcal{X}$-formulas.

The set of elementary formulas now contains: f; all $\mathcal{U}$-, $\mathcal{R}$-, $\mathcal{F}$-, $\mathcal{G}$-, and $\mathcal{GF}$-subformulas in the closure of f; as well as all subformulas g where $\mathcal{X}g$ is in the closure of f. Note that we treat the common $\mathcal{GF}$ combination as a single operator; a subformula of the form $\mathcal{GF}g$ adds a single element to the set of elementary formulas as opposed to two elements, one each for the $\mathcal{G}$- and $\mathcal{F}$-subformulas.

Again, we declare a Boolean SMV variable EL_g for every elementary formula g as well as Boolean variables for each atomic proposition in f. In addition, we declare a Boolean SMV *promise variable* P_g for every $\mathcal{U}$-, $\mathcal{F}$-, and $\mathcal{GF}$-subformula in the closure. These formulas are used to define fairness conditions. Intuitively, P_g holds when g is a promise for the future that is not yet fulfilled. If P_g does not hold, then the promise must be fulfilled immediately. To ensure satisfaction of eventualities we require that each promise variable P_g is false infinitely often. The TGBA encoding creates fewer EL variables than the CGH encoding, but it does add promise variables.

Again, we define an auxiliary variable S_h for every formula h in the closure of f. The characteristic function for S_h is defined similarly to the CGH encoding, with a few changes:

$$S_h = S_{g1} \& S_{g2} \text{ if } h = g_1 \wedge g_2$$

$$S_h = next(EL_g) \text{ if } h = \mathcal{X}g$$

$$S_h = S_{g2} | (S_{g1} \& P_{g1\,\mathcal{U}\,g2} \& (next(EL_{g1\,\mathcal{U}\,g2}))) \text{ if } h = g_1\,\mathcal{U}\,g_2$$

$$S_h = S_{g2} \& (S_{g1} | (next(EL_{g1\,\mathcal{R}\,g2}))) \text{ if } h = g_1\,\mathcal{R}\,g_2$$

$$S_h = S_g \& (next(EL_{\mathcal{G}\,g})) \text{ if } h = \mathcal{G}\,g$$

$$S_h = S_g | (P_{\mathcal{F}\,g} \& next(EL_{\mathcal{F}\,g})) \text{ if } h = \mathcal{F}\,g$$

$$S_h = (next(EL_{\mathcal{GF}\,g})) \& (S_g | P_{\mathcal{GF}\,g}) \text{ if } h = \mathcal{GF}\,g.$$

Since we reason directly over the temporal subformulas of f (and not over $\mathcal{X}g$ for temporal subformula g as in CGH), the transition relation associates elementary formulas with

matching elements of our characteristic function. Finally, we generate our symbolic TGBA; here is our SMV model M_f:

```
MODULE main

VAR
  /*declare a Boolean variable for each atomic prop in f*/
  a : boolean;
  ...

VAR /*declare a new variable for each elementary formula*/
  EL_f : boolean;      /*f is the input LTL formula*/
  EL_g1 : boolean;     /*g is an X-, F-, U-, or GF-formula*/
  ...
  EL_gn : boolean;

DEFINE /*characteristic function definition*/
   S_g = ...
   ...

TRANS /*for each EL-variable, generate a line here*/
  ( EL_g1 = S_g1 ) &   /*a line for every EL variable*/
  ...
  ( EL_gn = S_gn )

FAIRNESS (!P_g1) /*fairness constraint for each promise variable*/
...

FAIRNESS TRUE  /*only needed if there are no promise variables*/

SPEC !(EL_f & EG TRUE)
```

Symbolic TGBAs can only be created for NNF formulas because the model checker tries to guess a sequence of values for each of the promise variables to satisfy the subformulas, which does not work for negative $\mathcal{U}$-formulas. (This is also the case for explicit state model checking; SPOT also requires NNF for its TGBA encoding [130].) Consider the formula $f = \neg(a\ \mathcal{U}\ b)$ and the trace $\{$a=1,b=0$\}$, $\{$a=1,b=1$\}$, ... Clearly, $(a\ \mathcal{U}\ b)$ holds in the trace, so f fails in the trace. If, however, we chose P_aUb to be false at time 0, then EL_aUb is false at time 0, which means that f holds at time 0. The correctness

of our construction is summarized by the following theorem.

Theorem 5.3

Let M_f be the SMV program made by the TGBA encoding for LTL formula f. Then M_f does not satisfy the specification `!(EL_f & EG true)` *iff f is satisfiable.*

Proof: We prove each direction in turn.

Only if: If f is satisfiable then the specification `!(EL_f & EG true)` does not hold in M_f.

If f is satisfiable, there is a trace $\pi = \pi_0, \pi_1, \ldots \in (2^{Prop})^\omega$ such that $\pi, 0 \models f$ (trace π at time instant 0 satisfies f), where *Prop* is the set of atomic propositions occurring in f. To show that `!(EL_f & EG true)` does not hold in M_f, we need to exhibit an infinite trace π' of M_f such that EL_f holds at point 0 of π'. A trace π' of M_f is a trace $\pi' = \pi'_0, \pi'_1, \ldots \in (2^{Var(f)})^\omega$, where $Var(f)$ is the set of variables of M_f, consisting of:

- the atomic propositions *Prop*,

- the variable EL_g for each elementary formula g of f,

- and a promise variable P_g for each $\mathcal{U}, \mathcal{F}$, and $\mathcal{GF}$ subformula of f.

Note that $Prop \subseteq Var(f)$. We define π' as a conservative extension of π; that is, $\pi'_i \cap Prop = \pi_i$, for all $i \geq 0$. We define this extension as follows:

- for each elementary formula g of f, we have that $EL_g \in \pi'_i$ iff $\pi, i \models g$ (trace π at time instant $i \in \omega$ satisfies subformula g),

- for a subformula g of f, of the form $h'\mathcal{U}h, \mathcal{F}h$ or $\mathcal{GF}h$, we have that $P_g \in \pi'_i$ iff $\pi, i \models g$, but $\pi, i \not\models h$.

Note that since f is an elementary formula of itself and $\pi, 0 \models f$, we immediately have that $EL_f \in \pi'_0$.

It remains to show that π' is a trace of M_f. To that end we first extend π'. Let $Var'(f)$ be the set of *all* variables in M_f, including auxiliary variables. We define $\pi'' \in (2^{Var'(f)})^\omega$ as a conservative extension of π'; that is, $\pi''_i \cap Var(f) = \pi'_i$, for all $i \geq 0$. We define this extension as follows: for each subformula g of f, we have that $S_g \in \pi''_i$ iff $\pi, i \models g$.

We now need to show that all of the statements of M_f hold in π''. Each TRANS statement $EL_g = S_g$ holds trivially, as for each elementary formula g of f, we have that $EL_g \in \pi''_i$ iff $\pi, i \models g$ iff $S_g \in \pi''_i$. The DEFINE statements for $\wedge$, $\vee$, $\mathcal{X}$, $\mathcal{R}$, and $\mathcal{G}$ hold because of the basic properties of the propositional and temporal connectives [87]. For $\mathcal{F}$-, $\mathcal{U}$-, and $\mathcal{GF}$-subformulas, we also have to take into account promise variables. Consider, for example, a subformula h of the form $\mathcal{F}g$, for which the DEFINE statement is $S_h = S_g|(P_{\mathcal{F} g}\&next(EL_{\mathcal{F} g}))$. Suppose $S_h \in \pi''_i$, which means that $\pi, i \models \mathcal{F}g$. We know that $\pi, i \models \mathcal{F}g$ iff either $\pi, i \models g$ or both $\pi, i \not\models g$ and $\pi, i+1 \models h$. If $\pi, i \models g$, then $S_g \in \pi''_i$. If $\pi, i \not\models g$, then $P_{\mathcal{F}g} \in \pi''_i$ and $EL_h \in \pi''_{i+1}$. Conversely, if $S_g \in \pi''_i$, then $\pi, i \models g$, which entails $\pi, i \models \mathcal{F}g$ and, consequently, $S_h \in \pi''_i$. Also, if $EL_h \in \pi''_{i+1}$, then $\pi, i+1 \models h$, and, consequently, $\pi, i \models h$ and $S_h \in \pi''_i$. The arguments for $\mathcal{U}$- and $\mathcal{GF}$-subformulas are similar.

Finally, we need to show that the FAIRNESS statement holds. That is, for each promise variable P_h there are infinitely many i's such that $P_h \not\models \pi'_i$. Assume, for example, that h is the subformula $\mathcal{F}g$. Suppose to the contrary that the FAIRNESS statement fails. That is, there is some $i_0 \geq 0$ such that $P_h \in \pi'_i$ for all $i \geq i_0$. But this means that $\pi, i_0 \models \mathcal{F}g$, and $\pi, i \not\models g$ for all $i \geq i_0$, which is impossible. The arguments for $\mathcal{U}$- and GF-subformulas are similar.

If: If M_f does not satisfy the specification `!(EL_f & EG true)` then f is satisfiable.

Suppose the specification `!(EL_f & EG true)` does not hold in M_f. Then, by definition, there is an infinite trace $\pi' = \pi'_0, \pi'_1, \ldots \in (2^{Var(f)})^\omega$ of M_f such that EL_f holds at

point 0 of π', where $Var(f)$, as defined above, corresponds to the set of atomic propositions in f, elementary formula variables, and promise variables. To show that f is satisfiable, we show that the infinite trace $\pi = \pi_0, \pi_1, \ldots \in (2^{Prop})^\omega$, defined by $\pi_i = \pi'_i \cap Prop$ for $i \geq 0$, satisfies f at time 0; that is, $\pi, 0 \models f$.

Again, we first define $\pi'' \in (2^{Var'(f)})^\omega$ as a conservative extension of π', where $Var'(f)$ includes *all* variables in M_f, including auxiliary variables. The DEFINE statements define each auxiliary variable S_h in the characteristic function in terms of the set of variables in $Var(f)$, supplemented with an auxiliary variable S_g corresponding to each subformula g of h. Since π'' is defined over $Var'(f)$, the values for the auxiliary variables can be defined uniquely using the characteristic function.

We now prove that for every subformula h of f and $\forall i \geq 0$ we have that $S_h \in \pi''(i)$ entails $\pi, i \models h$. The proof is by induction on h.

Base case: For $h = p \in Prop$, we have that $S_h = p$. So $S_h \in \pi''(i)$ iff $p \in \pi(i)$ iff $\pi, i \models p$. If $h = \neg p$, then we have that $S_h = !p$. So $S_h \in \pi''(i)$ iff $S_p \notin \pi''(i)$ iff $\pi, i \not\models p$ iff $\pi, i \models h$,

Induction step: Assume the claim holds for subformulas g, g_1, g_2 of f.

- $h = g_1 \wedge g_2$.

  We have that $S_h = S_{g_1} \& S_{g_2}$. So $S_h \in \pi''(i)$ iff $S_{g_1} \in \pi''(i)$ and $S_{g_2} \in \pi''(i)$, which entails $\pi, i \models g_1$ and $\pi, i \models g_2$, which entails $\pi, i \models h$.

- $h = g_1 \vee g_2$.

  We have that $S_h = S_{g_1} \vee S_{g_2}$. So $S_h \in \pi''(i)$ iff $S_{g_1} \in \pi''(i)$ or $S_{g_2} \in \pi''(i)$, which entails $\pi, i \models g_1$ or $\pi, i \models g_2$, which entails $\pi, i \models h$.

- $h = \mathcal{X}g$.

  We have that $S_h = next(EL\_g)$. So $S_h \in \pi''(h)$ iff $EL_g \in \pi''(i + 1)$ iff (by the TRANS

statement) $S_g \in \pi''(i + 1)$, which, by induction, entails $\pi, i + 1 \models g$, which entails $\pi, i \models h$.

- $h = g_1 \, \mathcal{U} \, g_2$.

  We have that $S_h = S_{g_2} | (S_{g_1} \& P_h \& (next(EL_h)))$. Suppose $S_h \in \pi''(i)$. Then either $S_{g_2} \in \pi''(i)$ or $S_{g_1} \in \pi''(i)$, $P_h \in \pi''(i)$, and $EL_h \in \pi''(i + 1)$. Note that, by the TRANS statement, $EL_h \in \pi''(i + 1)$ iff $S_h \in \pi''(i + 1)$. Thus, S_{g_2} can be "postponed," at the cost of maintaining S_{g_1} and P_h. But the FAIRNESS statement implies that there is some $j : j \geq i$ where $P_h \notin \pi''(j)$. Choose the smallest such j. Then we have that $S_{g_2} \in \pi''(j)$, and $\forall k : i \leq k < j$, we have that $S_{g_1} \in \pi''(k)$. By induction, $\pi, j \models g_2$, and $\forall k, i \leq k < j$, we have that $\pi, k \models g_1$. It follows that $\pi, i \models h$.

- $h = g_1 \, \mathcal{R} \, g_2$.

  We have that $S_h = S_{g_2} \& (S_{g_1} | (next(EL_h)))$. So $S_h \in \pi''(i)$ iff $S_{g_2} \in \pi''(i)$ and either $S_{g_1} \in \pi''(i)$ or $EL_h \in \pi''(i + 1)$. Note that, by the TRANS statement, $EL_h \in \pi''(i + 1)$ iff $S_h \in \pi''(i + 1)$. It follows that S_{g_2} is "propagated" until "released" by S_{g1}. That is, if $S_h \in \pi''(i)$ then $\forall j : j \geq i$, either $S_{g_2} \in \pi''(j)$ or there is some $k : i \leq k < j$, such that $S_{g1} \in \pi''(k)$. By induction, for all $j : j \geq i$, either $\pi, j \models g_2$ or there is some $k : i \leq k < j$, such that $\pi, k \models g_1$. It follows that $\pi, i \models h$.

- $h = \mathcal{G}g$.

  We have that $S_h = S_g \& (next(EL_h))$. So $S_h \in \pi''(i)$ iff $S_g \in \pi''(i)$ and $EL_h \in \pi''(i + 1)$ iff $g \in \pi(i)$ and (by the TRANS statement) $S_{\mathcal{G}g} \in \pi''(i + 1)$. It follows that S_g is continually propagated. That is, if $S_h \in \pi''(i)$, then, for all $j : j \geq i$, we have that $S_g \in \pi''(j)$. By induction, for all $j : j \geq i$, we have that $\pi, j \models g$. It follows that $\pi, i \models h$.

- $h = \mathcal{F}g$.

We have that $S_h = S_g|(P_h \& next(EL_h))$. Suppose $S_h \in \pi''(i)$. Then either $S_g \in \pi''(i)$ or $P_h \in \pi''(i)$ and $EL_h \in \pi''(i+1)$. Note that, by the TRANS statement, $EL_h \in \pi''(i+1)$ iff $S_h \in \pi''(i+1)$. Thus, S_g can be "postponed," at the cost of maintaining P_h. But the FAIRNESS statement implies that there is some $j : j \geq i$ where $P_h \notin \pi''(j)$. Choose the smallest such j. Then we have that $S_g \in \pi''(j)$. By induction, $\pi, j \models g$. It follows that $\pi, i \models h$.

- $h = \mathcal{GF}g$.

  We have that $S_h = ((next(EL_h)) \& (S_g|P_h)$. Suppose $S_h \in \pi''(i)$. Then $EL_h \in \pi''(i + 1)$ and either $S_g \in \pi''(i)$ or $P_h \in \pi''(i)$. Note that, by the TRANS statement, $EL_h \in \pi''(i + 1)$ iff $S_h \in \pi''(i + 1)$. Thus, for all $j : j \geq i$ we have that $S_h \in \pi''(j)$. Again, we note that S_g can be "postponed" at the cost of maintaining P_h. But the FAIRNESS statement implies that for every $j : j \geq i$ there is some $k : k \geq j$ where $P_h \notin \pi''(k)$. For each $j \geq i$, choose smallest such k; call it k_j. Then we have that $S_g \in \pi''(k_j)$. By induction, $\pi, k_j \models g$. It follows that $\pi, i \models h$.

By assumption $EL_f \in \pi''(0)$. By the TRANS statement it follows that $S_f \in \pi''(0)$, and therefore $\pi, 0 \models f$. Therefore, f is satisfiable.

5.4   A Set of 30 Symbolic Automata Encodings

Our novel encodings are combinations of four components: (1) Normal Form: BNF or NNF, described above, (2) Automaton Form: GBA or TGBA, described above, (3) Transition Form: fussy or sloppy, described below, and (4) Variable Order: default, naïve, LEXP, LEXM, MCS-MIN, MCS-MAX, described below. In total, we have 30 novel encodings, since BNF can only be used with fussy-encoded GBAs, as explained below. The CGH encoding corresponds to BNF/fussy/GBA; we encode this combination with all six variable orders.

Automaton Form   As discussed earlier, CGH is based on GBA, in combination with BNF. We can combine, however, GBA also with NNF. For this, we need to expand the characteristic function for symbolic GBAs in order to form them from NNF formulas:

$$S_h = S_{g1} \& S_{g2} \text{ if } h = g_1 \wedge g_2 \qquad\qquad S_h = S_g \& S_{\chi_{(\mathcal{G}g)}} \text{ if } h = \mathcal{G}g$$

$$S_h = S_{g2} \& (S_{g1} | S_{\chi_{(g1 \, \mathcal{R} \, g2)}}) \text{ if } h = g_1 \, \mathcal{R} \, g_2 \qquad\qquad S_h = S_g | S_{\chi_{(\mathcal{F}g)}} \text{ if } h = \mathcal{F}g$$

Since our focus here is on symbolic encoding, PANDA, unlike CadenceSMV, does not apply formula rewriting and related optimizations; rather, PANDA's symbolic automata are created directly from the given normal form of the formula. Formula rewriting may lead to further improvement in PANDA's performance.

Sloppy Encoding: A Novel Transition Form   CGH employs iff-transitions, of the form `TRANS (EL_g=(S_g))`. We refer to this as *fussy* encoding. For formulas in NNF, we can use only-if transitions of the form `TRANS (EL_g->(S_g))`, which we refer to as *sloppy* encoding. A similar idea was shown to be useful in the context of modal satisfiability solving [197]. Sloppy encoding increases the level of nondeterminism, yielding a looser, less constrained encoding of symbolic automata, which in many cases results in smaller BDDs. A side-by-side example of the differences between GBA and TGBA encodings (demonstrating the sloppy transition form) for formula $f = ((\mathcal{X}a)\&(b \, \mathcal{U} \, (!a)))$ is given in Figures 5.1-5.2.

A New Way of Choosing BDD Variable Orders   Symbolic model checkers search for a fair trace in the model-automaton product using a BDD-based fixpoint algorithm, a process whose efficacy is highly sensitive to variable order [4]. Finding an optimal BDD variable order is NP-hard, and good heuristics for variable ordering are crucial; see Chapter 2.9.1.

Recall that we define state variables in the symbolic model for only certain subformulas: $p \in AP$, EL_g, and P_g for some subformulas g. We form the variable graph by

```
MODULE main
/*formula: ((X (a )) & ((b )U (!(a ))))*/
VAR /*a Boolean var for each prop in f*/
  a : boolean;
  b : boolean;
VAR /*a var EL_X_g for each formula (X g) in
     el_list w/primary op X, U, R, G, or F*/
  EL_X_a : boolean;
  EL_X__b_U_NOT_a : boolean;
DEFINE
/*each S_h in the characteristic function*/
  S__X_a__AND__b_U_NOT_a :=
    (EL_X_a) & (S__b_U_NOT_a);
  S__b_U_NOT_a :=
    (!(a )) | (b & EL_X__b_U_NOT_a);


TRANS /*a line for each (X g) in el_list*/
  ( EL_X_a -> (next(a) ) ) &
  ( EL_X__b_U_NOT_a -> (next(S__b_U_NOT_a) ))

FAIRNESS (!S__b_U_NOT_a | (!(a )))
SPEC    !(S__X_a__AND__b_U_NOT_a & EG TRUE)
```

```
MODULE main
/*formula: ((X (a ))& ((b )U (!(a ))))*/
VAR /*a Boolean var for each prop in f*/
  a : boolean;
  b : boolean;
VAR /*a var for each EL_var in el_list*/
  EL__X_a__AND__b_U_NOT_a : boolean;
  P__b_U_NOT_a: boolean;
  EL__b_U_NOT_a : boolean;
DEFINE
/*each S_h in the characteristic function*/
  S__X_a__AND__b_U_NOT_a :=
    (S_X_a) & (EL__b_U_NOT_a);
  S_X_a := (next(a));
  S__b_U_NOT_a := ( ((!(a )))
    | (b & P__b_U_NOT_a
      & (next(EL__b_U_NOT_a)))));
TRANS /*a line for each EL_var in el_list*/
  ( EL__X_a__AND__b_U_NOT_a ->
    (S__X_a__AND__b_U_NOT_a) ) &
  ( EL__b_U_NOT_a -> (S__b_U_NOT_a) )
FAIRNESS (!P__b_U_NOT_a)
SPEC    !(EL__X_a__AND__b_U_NOT_a & EG TRUE)
```

Figure 5.1 : NNF/sloppy/GBA encoding for CadenceSMV.

Figure 5.2 : NNF/sloppy/TGBA encoding for CadenceSMV.

(a) GBA variable graph.

(b) TGBA variable graph.
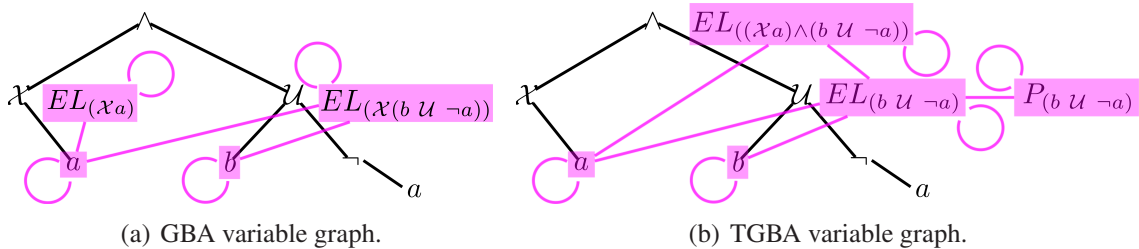
Figure 5.3 : The variable graphs in (a) and (b) were both formed from the parse tree for $f = ((\mathcal{X}a) \wedge (b \, \mathcal{U} \, \neg a))$.

identifying nodes in the input formula parse tree that correspond to the primary operators of those subformulas. Since we declare different variables for the GBA and TGBA encodings, the variable graph for a formula f may vary depending on the automaton form we choose. Figure 5.3 displays the GBA and TGBA variable graphs for an example formula, overlaid on the parse tree for this formula. We connect each variable-labeled vertex to its closest variable-labeled vertex descendant(s), skipping over vertices in the parse tree that do not correspond to state variables in our automaton construction. We create one node

per subformula variable, irrespective of the number of occurrences of the subformula; for example, we create only one node for the proposition *a* in Figure 5.3.

We implement five variable-ordering schemes, all of which take the variable graph as input. We compare these to the *default* heuristic of CadenceSMV. The *naïve* variable order is formed directly from a pre-order, depth-first traversal of the variable graph. We derive four additional variable-ordering heuristics by repurposing node-ordering algorithms designed for graph triangulation [209].[2] We use two variants of a lexicographic breadth-first search algorithm: variants *perfect* (LEXP) and *minimal* (LEXM). LEXP labels each vertex in the variable graph with its already-ordered neighbors; the unordered vertex with the lexicographically largest label is selected next in the variable order. LEXM operates similarly, but labels unordered vertices with both their neighbors and also all vertices that can be reached by a path of unordered vertices with smaller labels. The maximum-cardinality search (MCS) variable-ordering scheme differs in the vertex selection criterion, selecting the vertex in the variable graph adjacent to the highest number of already ordered vertices next. We seed MCS with an initial vertex, chosen either to have the *maximum* (MCS-MAX) or *minimum* (MCS-MIN) degree.

5.5 Experimental Methodology

Test Methods Each test was performed in two steps. First, we applied our symbolic encodings to the input formula. Second, each symbolic automaton and variable order file pair was checked by CadenceSMV. Since encoding time is minimal and heavily dominated by model-analysis time (the time to check the model for nonemptiness to determine LTL satisfiability) we focus exclusively on the latter here.

---

[2]The graph triangulation implementation we used was coded by the Kavraki Lab at Rice University.

Platform   We ran all tests on Shared University Grid at Rice (SUG@R), an Intel Xeon compute cluster.[3] SUG@R is comprised of 134 SunFire x4150 nodes, each with two quad-core Intel Xeon processors running at 2.83GHz and 16GB of RAM per processor. The OS is Red Hat Enterprise 5 Linux, 2.6.18 kernel. Each test was run with exclusive access to one node. Times were measured using the Unix `time` command. More details on the code used for the experiments described here are available in Appendix B and online.

Input Formulas   We employed a widely-used [56, 57, 206, 58] collection of benchmark formulas, established by [56]. All encodings were tested using three types of scalable formulas: random, counter, and pattern. Definitions of these formulas are given in Chapter 3. Our test set includes four counter and nine pattern formula variations, each of which scales to a large number of variables, and 60,000 random formulas, as detailed in Chapter 3.2, 3.3, and 3.4, respectively.

Correctness   In addition to proving the correctness of our algorithm, the correctness of our implementation was established by comparing for every formula in our large benchmark suite, the results (either SAT or UNSAT) returned by all encodings studied here, as well as the results returned by CadenceSMV for checking the same formula as an LTL specification for the universal model given in Chapter 3.5.2. We never encountered an inconsistency.

5.6   Experimental Results

Our experiments demonstrate that the novel encoding methods we have introduced significantly improve the translation of LTL formulas to symbolic automata, as measured in time to check the resulting automata for nonemptiness and the size of the state space we

---

[3]`http://rcsg.rice.edu/sugar/`

can check. No single encoding, however, consistently dominates for all types of formulas. Instead, we find that different encodings are better suited to different formulas. Therefore, we recommend using a multi-encoding approach, a variant of the multi-engine approach [210], of running all encodings in parallel and terminating when the first job completes.

Seven configurations are not competitive   While we can not predict the best encodings, we can reliably predict the worst. The following encodings are never optimal for any formulas in our test set. Thus, out of our 30 possible encodings, we rule out these seven:

- BNF/fussy/GBA/LEXM (essentially CGH with LEXM)

- NNF/fussy/GBA/LEXM
- NNF/fussy/TGBA/MCS-MAX

- NNF/fussy/TGBA/LEXM
- NNF/sloppy/TGBA/MCS-MAX

- NNF/sloppy/GBA/LEXM
- NNF/sloppy/TGBA/MCS-MIN

NNF is the best normal form, most (but not all) of the time.   NNF encodings are always better for all counter and pattern formulas; see, for example, Figure 5.4. Figure 5.5 demonstrates the use of both normal forms in the optimal encodings chosen by PANDA for random formulas. BNF encodings are occasionally significantly better than NNF; the solid blue point in Figure 5.5 corresponds to a formula for which PANDA's best BNF encoding is more than four times faster than the best NNF encoding. NNF is best much more often than BNF, likely because using NNF has the added benefit that it allows us to employ our sloppy encoding and TGBAs, which often carry their own performance advantages.

No automaton form is best.   Our TGBA encodings dominate for R_2, S, and U pattern formulas, and both types of 3-variable counter formulas. For instance, the log-scale plot in Figure 5.6 shows that the median model analysis time for R_2 pattern formulas encoded
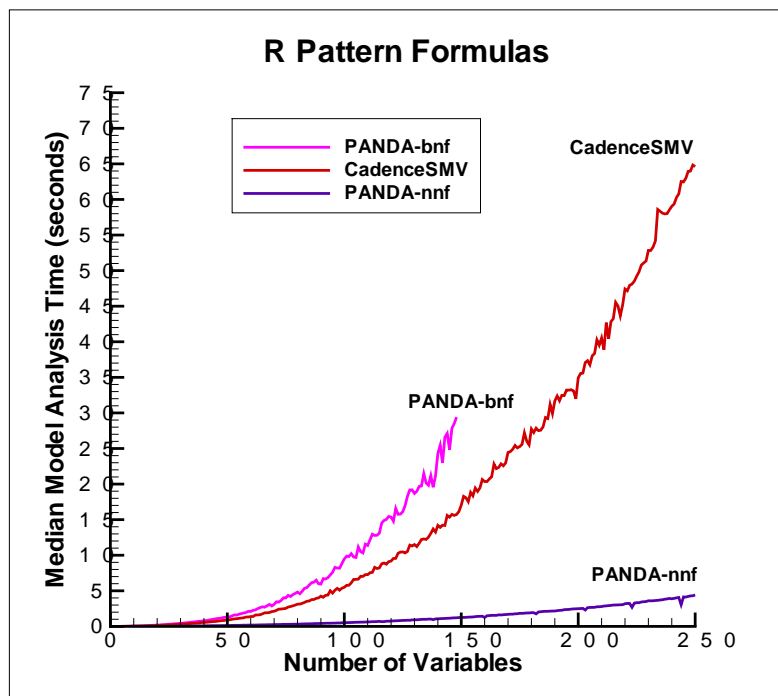
Figure 5.4 : Median model analysis time for $R(n) = \bigwedge_{i=1}^{n} (\mathcal{GF}p_i \vee \mathcal{FG}p_{i+1})$ for PANDA NNF/sloppy/GBA/`naïve`, CadenceSMV, and the best BNF encoding.
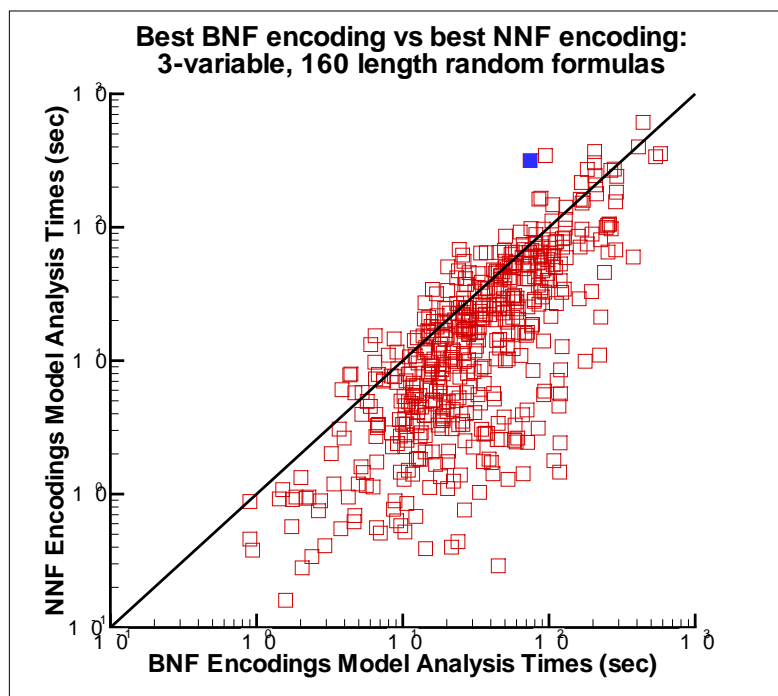
Figure 5.5 : Best encodings of 500 3-variable, 160 length random formulas. Points fall below the diagonal when NNF is better.

using PANDA's TGBA form grows subexponentially as a function of the number of variables, while CadenceSMV's median model analysis time for the same formulas grows exponentially. (The best of PANDA's GBA encodings is also graphed for comparison.) GBA encodings are better for other pattern formulas, both types of 2-variable counter formulas, and the majority of random formulas; Figure 5.7 demonstrates this trend for 180 length random formulas.

No transition form is best   Sloppy is the best transition form for all pattern formulas. For instance, the log-scale plot of Figure 5.8 illustrates that the median model analysis time for U pattern formulas encoded with PANDA's sloppy transition form grows subexponentially as a function of the number of variables, while CadenceSMV's median model analysis time for the same formulas grows exponentially. Fussy encoding is better for all counter formulas. The best encodings of random formulas are split between fussy and sloppy. Figure 5.9 demonstrates this trend for 140 length random formulas.

No variable order is best, but `LEXM` is worst.   The best encodings for our benchmark formula set are split between five variable orders. The naïve and default orders are optimal for more random formulas than the other orders. Figure 5.10 demonstrates that neither the naïve order nor the default order is better than the other for random formulas. The naïve order is optimal for E, Q, R, U_2, and S patterns. `MCS-MAX` is optimal for 2- and 3-variable linear counters. The `LEXP` variable order dominates for C_1, C_2, U, and R_2 pattern formulas, as well as for 2- and 3-variable counter formulas, yet it is rarely best for random formulas. Figure 5.11 demonstrates the marked difference in scalability provided by using PANDA with the `LEXP` order over running CadenceSMV on 3-variable counter formulas. We can analyze much larger models with PANDA using `LEXP` than with the native CadenceSMV encoding before memory-out. We never found the `LEXM` order to be the single best encoding
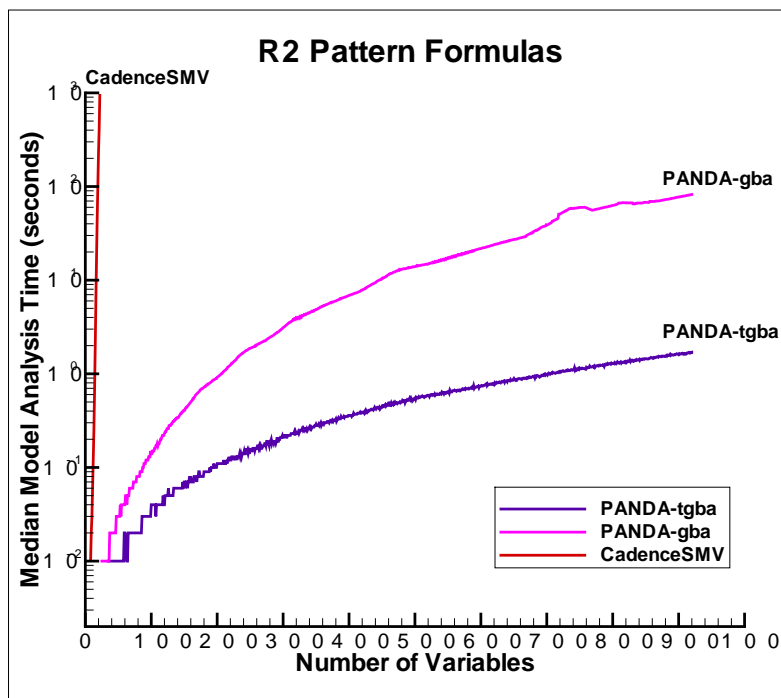
Figure 5.6 : $R_2(n) = (..(p_1 \mathcal{R} p_2) \mathcal{R} \ldots) \mathcal{R} p_n$. PANDA's NNF/sloppy/TGBA/LEXP encoding scales better than the best GBA encoding, NNF/sloppy/GBA/naïve, and exponentially better than CadenceSMV.
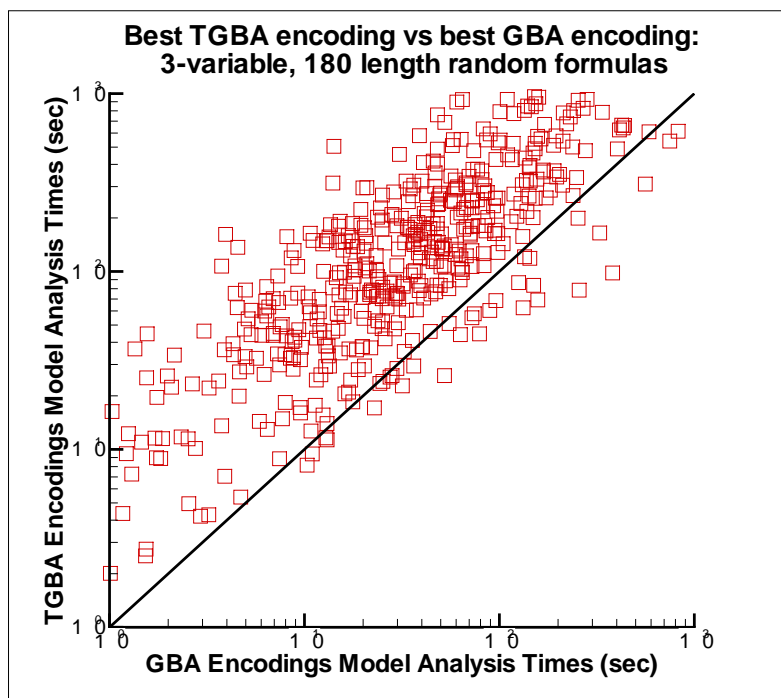
Figure 5.7 : Best encodings of 500 3-variable, 180 length random formulas. Points fall above the diagonal when GBA is better
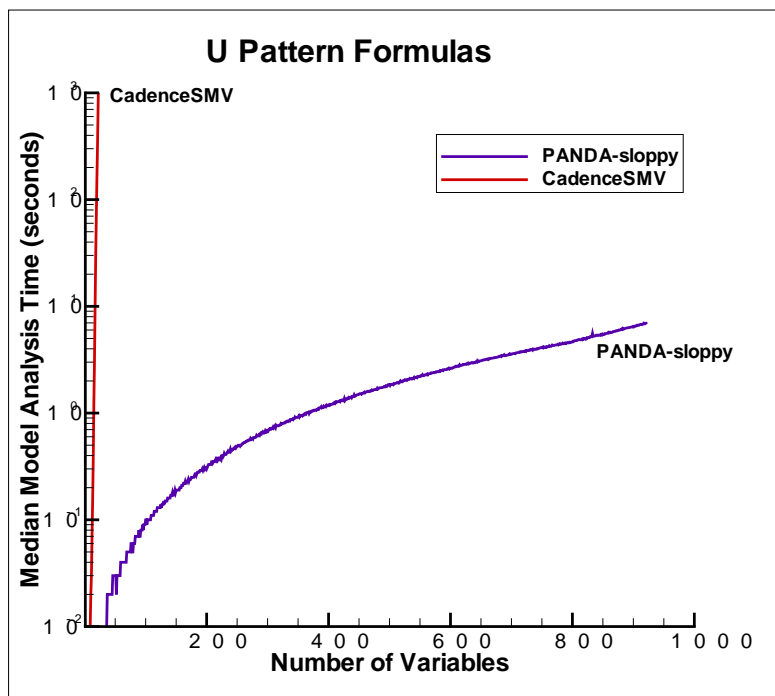
Figure 5.8 : $U(n) = (\dots(p_1\ \mathcal{U}\ p_2)\ \mathcal{U}\ \dots)\ \mathcal{U}\ p_n$. PANDA's NNF/sloppy/TGBA/LEXP scales exponentially better than CadenceSMV.
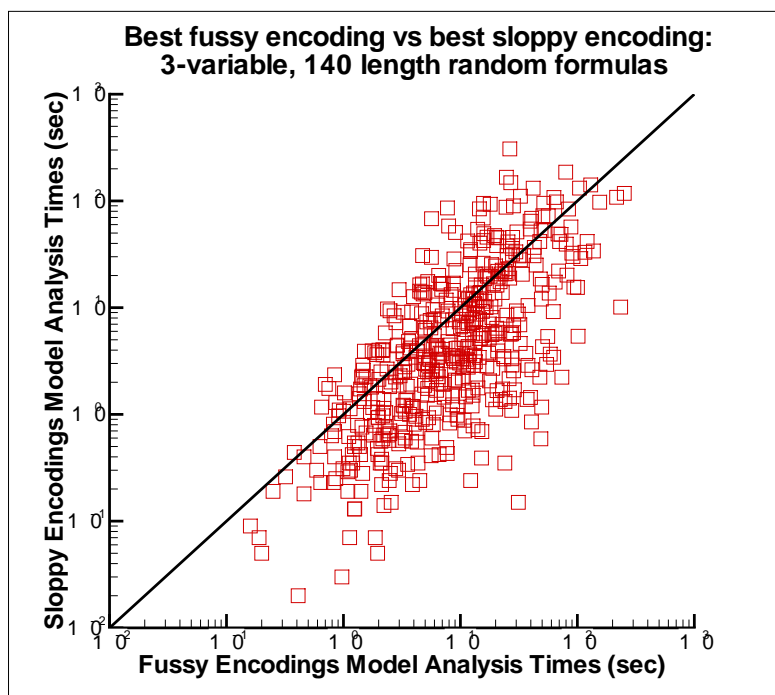
Figure 5.9 : Best encodings of 500 3-variable, 140 length random formulas. Points fall below the diagonal when sloppy encoding is best.

for any formula.

A formula class typically has a best encoding, but predictions are difficult   While each of our pattern and counter formulas has a best (or a pair of best) encodings, which remain consistent as we scale the formulas, we found that we could not reliably predict the best encoding using any statistics gathered from parsing, such as operator counts or ratios, levels of nesting, number of variables, etc. For example, we found that the best encoding for a pattern formula was not necessarily the best for a randomly-generated formula comprised of the same temporal operators. We surmise that the best encoding is tied to the structure of the formula on a deeper level; developing an accurate heuristic is left to future work.

There is no single best encoding; a multi-encoding approach is clearly superior   We implement a novel multi-encoding approach: our new PANDA tool creates several encodings of a formula and uses a symbolic model checker to check them for satisfiability in parallel, terminating when the first check completes. Our experimental data supports this multi-encoding approach. Figures 5.4, 5.6, and 5.8 highlight the significant decrease in CadenceSMV model analysis time for R, R_2, and U pattern formulas, while Figure 5.11 demonstrates increased scalability in terms of state space using counter formulas. Altogether, we demonstrate that a multi-encoding approach is dramatically more scalable than the current state of the art. The increase in scalability is dependant on the specific formula, though for some formulas PANDA's model analysis time is exponentially better than CadenceSMV's model analysis time for the same class of formulas.

5.6.1   Application Benchmarks

In order to demonstrate further that our PANDA encoding outperforms the native encoding of CadenceSMV for real-life LTL satisfiability checking, we also tested both tools on a

**Best encodings with naive vs default variable orders
3-variable, 195 length random formulas**
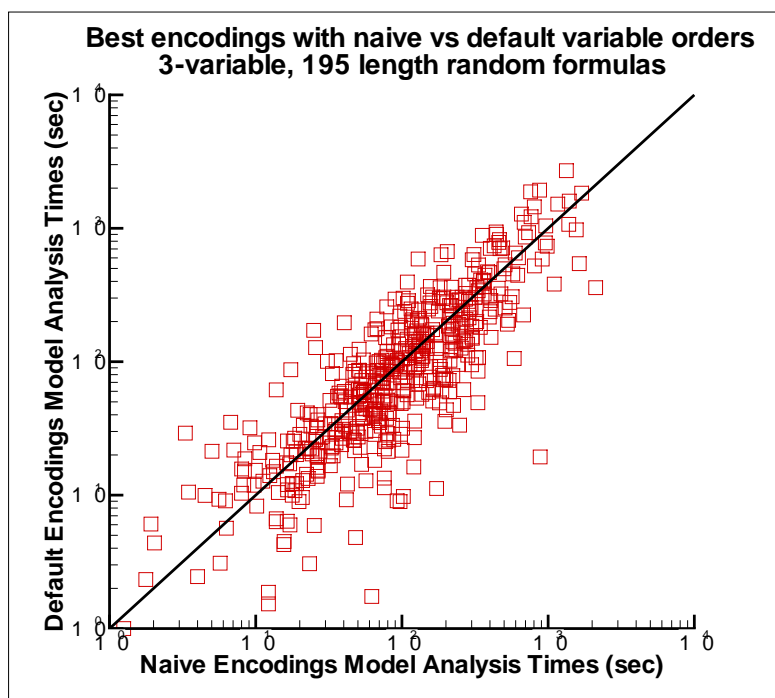
Figure 5.10 : Best encodings of 500 3-variable, 195 length random formulas. Points fall above the diagonal when naïve variable order is best.

3-variable Counter Formulas
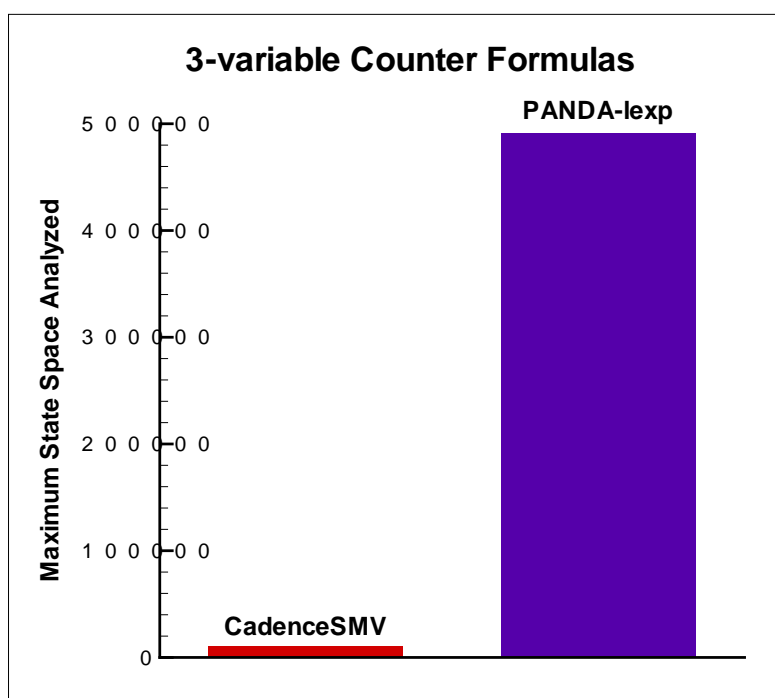
Figure 5.11 : Maximum states analyzed before space-out. CadenceSMV quits at 10240 states. PANDA's NNF/fussy/TGBA/LEXP scales to 491520 states.

set of application benchmarks, comprised of formulas used to specify actual systems. Our application benchmark formulas come from six sources:[4]

1. **acacia demo-v22:** 10 formulas

2. **acacia demo-v3:** 6 formulas

3. **acacia example:** 25 formulas

4. **alaska szymanski:** 4 formulas

5. **anzu amba:** 8 formulas

6. **anzu genbuf:** 10 formulas

The *acacia demo-v22*, *acacia demo-v3*, and *acacia example* formulas are specifications for systems such as arbiters and traffic-light controllers, distributed with the Acacia tool,[5] as developed for a study on LTL realizability and synthesis [211]. The *alaska szymanski* formulas[6] were developed as liveness properties for the Szymanski mutual exclusion protocol for LTL satisfiability and model checking [57]. The Anzu[7] benchmarks are sets of formulas used for synthesizing industrial hardware systems from specifications, combined into monolithic formulas for the purpose of satisfiability checking [212]. The *anzu amba* formulas are specifications for advanced microcontroller bus architectures while the *anzu genbuf* specifications describe generalized buffers.

We applied PANDA and CadenceSMV to these 63 application benchmark formulas. PANDA completed 51 formulas before spacing out, while CadenceSMV completed 45

---

[4]Thanks to Viktor Schuppan for suggesting these sources, providing some of the formulas in SMV format, and constructing the Anzu formula combinations.

[5]http://www.antichains.be/acacia/src/acacia_9_linux_i386.tar.gz

[6]http://www.antichains.be/alaska/tacas08_experiments.zip

[7]http://www.iaik.tugraz.at/content/research/design_verification/anzu/
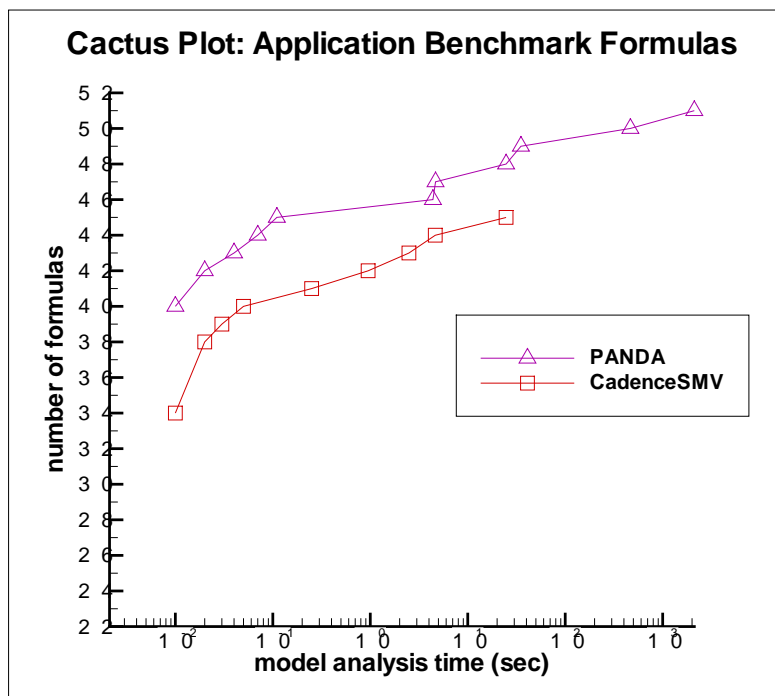
Figure 5.12 : Cactus plot: median model analysis time over all application benchmarks for CadenceSMV and the best PANDA encoding.

formulas before spacing out. The comparison of the performance of CadenceSMV and PANDA's best encoding on these application benchmark formulas is plotted in Figure 5.12 using a classical cactus plot: the y-axis shows how many instances were checked in time less than or equal to the runtime given on the x-axis, presuming they are run in parallel. PANDA solved more formulas and did it in less time than CadenceSMV.

5.7   Discussion

This chapter brought attention to the issue of scalable construction of symbolic automata for LTL formulas in the context of LTL satisfiability checking. We defined novel encodings and novel BDD variable orders for accomplishing this task. We explored the impact of these encodings, comprised of combinations of normal forms, automaton forms, transition forms,

and variable orders. We showed that each can have a significant impact on performance. At the same time, we showed that no single encoding outperforms all others and showed that a multi-encoding approach yields the best result, consistently outperforming the native translation of CadenceSMV.

We do not claim to have exhaustively covered the space of possible encodings of symbolic automata. Several papers on the automata-theoretic approach to LTL describe approaches that could be turned into alternative encodings of symbolic automata, cf. [137, 213, 214, 215]. The advantage of the multi-encoding approach we introduced here is its *extensibility*; adding additional encodings is straightforward. The multi-encoding approach can also be combined with different back ends. In this chapter we used CadenceSMV as a BDD-based back end; using another symbolic back end (cf. [57]) or a SAT-based back end (cf. [216]) would be an alternative approach, as both BDD-based and SAT-based back ends require symbolic automata. Since LTL serves as the basis for industrial languages such as PSL and SVA, the encoding techniques studied here may also serve as the basis for novel encodings of such languages, cf. [217, 204].

While a thorough investigation led us to conclude it is not possible to predict the best encoding for a given formula directly from any statistics that can be gathered from the formula during parse time, this does not exclude the possibility that there may be a different way to accomplish this task. It may be possible to investigate how different types of structural properties of specification formulas and techniques for encoding them are related to the efficiency of satisfiability solving or model checking those encodings, similar to the kind of structural analysis that has been accomplished in the propositional satisfiability community [218]. It may be possible to use sophisticated machine learning techniques to downselect from the possible encodings a smaller subset predicted to achieve optimal performance [219, 220]. A similar investigation has not yet been done work in the domain of symbolic LTL-to-automata but it is a possible direction for future work. Note that com-

petitive parallel execution strategies similar to the approach presented here are also used to achieve speedups in the propositional satisfiability domain in cases where it is not clear which compilation of a formula will perform best [221].

In this chapter we examined our novel symbolic encodings of LTL in the context of satisfiability checking. An important difference between satisfiability checking and model checking is that in the former we expect to have to handle much larger formulas, since we need to consider the conjunction of properties. Also, in model checking, the size of the symbolic automata can be dwarfed by the size of the model under verification. Thus, the issue of symbolic encoding of automata in the context of model checking deserves a separate investigation.

Chapter 6

Improved Algorithm for Explicit LTL Satisfiability and Model Checking

6.1 Introduction

The translation of LTL specifications constitutes an essential step in model checking and a major influence on the efficiency of formal verification via model checking. Recall that in model checking, the negation of the specification is translated into a Büchi automaton, combined with the system model, and then checked for nonemptiness [39]. The model checker searches for a *counterexample* in the form of an infinite fair trace in this combined model that is shaped like a lasso, starting from an initial system state and terminating in an infinite loop where the system violates its specification. The encoding of the LTL specification is an important consideration for efficient model checking as it influences the efficiency of this search. An extensive survey of LTL-to-automaton translation algorithms demonstrated that the difference in performance between LTL-to-automaton translators can be dramatic, having a major impact on the performance of both symbolic and explicit-state model checking [56]. In this chapter, we devise a new explicit-state translation of LTL-to-automata that consistently results in better model checking performance, for a large array of benchmarks, over the best current translation for the class of LTL specifications that describe *safety properties*.

Safety properties are arguably the most used formal specifications in practice, encapsulating the desired behaviors of a wide variety of real-world systems. Many applications of fault tolerance [222], hardware resets (which entail truncating the execution path

and canceling future obligations) [223], dynamic verification applications such as monitoring assertion failures [224], and other applications in runtime verification [225] constitute safety properties. They can describe most intended properties of real-time systems, since responses are required within bounded intervals [226], and are integral in the assume-guarantee paradigm, where general verification methods are improved if either the assumption or the guarantee is safe [144].

A *safety property* expresses the sentiment that "something bad never happens." Intuitively, "something bad" only needs to happen once in a computation for the property to be violated; what happens after that in the infinite computation does not affect the outcome of the model-checking step. In this sense, a violation of a safety property can always be witnessed by a finite prefix. For property φ and infinite computation $\pi = \pi_0, \pi_1, \ldots$, φ is a safety property satisfied by computation π if and only if for all lengths of finite prefixes of π, that finite prefix concatenated with some infinite computation models φ. Oppositely, if $\pi \not\models \varphi$ then there is some finite prefix of π, in which something bad happens, that cannot be extended into an infinite computation that will satisfy φ. This property allows us to utilize finite automata, instead of Büchi automata, for model checking a safety property φ. We can build an automaton that accepts only the bad prefixes of the safety formula and returns a finite error trace [144]. Safety specifications allow us to take advantage of *determinism* in LTL-to-automaton translation.

An automaton is *deterministic* when there is exactly one transition from every state for every input character. When we combine a deterministic automaton representing the behavior specification with the system model, the ensuing nonemptiness check is computationally simpler than if the specification automaton is nondeterministic, perhaps even dramatically so. A deterministic specification automaton can have a shorter model checking time than a nondeterministic one because it causes less branching in the product automaton when the specification and system model automata are combined. Intuitively, when the specification

automaton is nondeterministic, the model checker has to search for a counterexample trace both in the system and in the specification automata. If the automaton is deterministic, then there is no need in the second search since for every trace of the system model there is a unique run of the specification automaton. Note that determinization of finite automata for safety properties is much less complex than determinization of ω-automata [227]. While for any nondeterministic finite automaton with n states there is an equivalent deterministic automaton with $2^{O(n)}$ states, it was proved in [151] that the optimal bound for ω-automata is $2^{O(n \log(n))}$.

A deterministic automaton can be much larger than an equivalent nondeterministic automaton; however, it was previously shown that automaton size is not the primary determining factor in the efficiency of the nonemptiness check [228]. Indeed larger, more deterministic automata have resulted in smaller product automata and more efficient model-checking times [139, 229, 64]. Determinism is actually required if the automata are to be used for monitoring executions of software [189, 1]. Similar questions have been explored for the past-time variant of LTL, which is theoretically as expressive as the standard, future-time LTL presented here, but much more natural to determinize [226, 213, 222].

We introduce 26 novel encodings of LTL safety properties as deterministic automata in the form of Promela (PROcess MEta LAnguage) `never` claims for the explicit model checker Spin [59]. We implement these encodings as an extension of the open-source CHIMP tool[1] [1] that creates SystemC monitors for LTL formulas; we call our extension CHIMP-Spin since we are creating Spin `never` claims instead. We compare our model checking performance against SPOT [140] since [228] found that SPOT outperforms all other explicit LTL-to-automaton translators. We demonstrate over a large array of benchmarks that using one of our encodings for model checking of safety properties consistently

---

[1]`http://sourceforge.net/projects/chimp-rice/`

results in better model checking times than using the SPOT encoding.

A key point of our construction is that we concentrate on reducing *model checking time*. Since in real-world applications of model checking, specifications are written once and checked against a changing system design multiple times, we do not focus on LTL-to-automaton translation or specification compilation times. This is particularly pertinent for regression testing: when the system is changed to fix a bug or add a new feature it is necessary to re-check all properties to ensure previous tests produce the same results. Figure 6.1 depicts the Spin model checking process; we measure compilation of LTL-to-`never` claim, `never` claim-to-C, and C-to-binary separately from model checking execution time. Because we run SPOT as a step in the creation of each of our new encodings, the specification automaton generation times incurred by our algorithm will always be greater than running SPOT alone. (It is important to note that our automaton generation times are consistently dwarfed by the corresponding model checking times.) To streamline regression testing, we argue that future versions of Spin should not require us to recompile `never` claims for each run of the model checker, even when they have not changed. Such an adjustment would more accurately reflect industrial applications of model checking and, combined with our reduced model checking times, reduce the amortized cost of model checking.
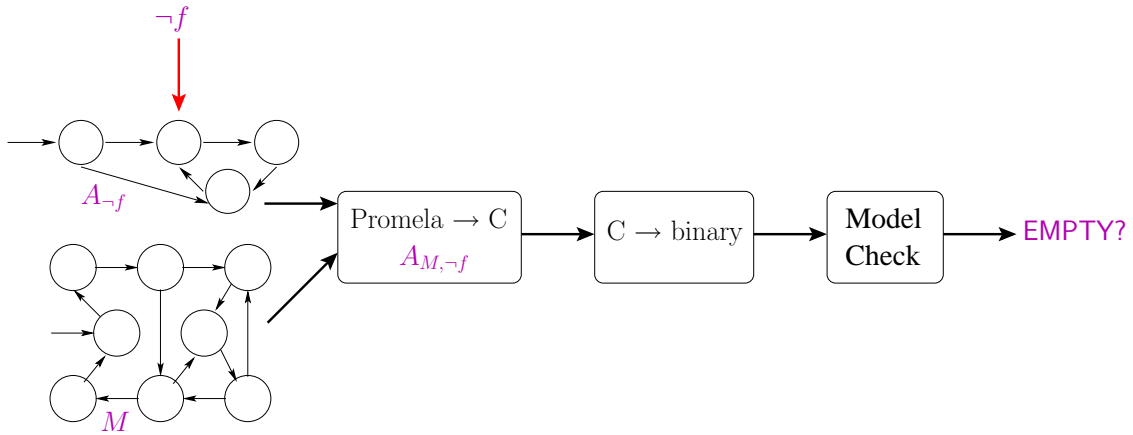
Figure 6.1 : System Diagram illustrating the Spin model checking process.

The structure of this chapter is as follows. We detail the theory underlying our construction of deterministic encodings of LTL safety specifications in Section 6.2 and demonstrate our 26 novel constructions of Promela `never` claims in Section 6.3. We describe our experimental methods in Section 6.4, present our experimental results and demonstrate that we can consistently out-perform the current best algorithm for LTL-to-automata, SPOT, in Section 6.5, and conclude with a discussion in Section 6.6.

6.2 Theoretical Background

Recall from Chapter 2.4 that we can construct an *Nondeterministic Büchi Automaton* NBA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ from any LTL formula φ. In the automata-theoretic approach to model checking [39] we negate the LTL formula φ, translate $\neg \varphi$ into the automaton $\mathcal{A}_{\neg \varphi}$ and compose $\mathcal{A}_{\neg \varphi}$ with the model M under verification, forming the automaton $\mathcal{A}_{M, \neg \varphi}$, which the model checker searches for nonemptiness. If there is no accepting run of $\mathcal{A}_{M, \neg \varphi}$ (i.e. the language $\mathscr{L}(\mathcal{A}_{M, \neg \varphi}) = \emptyset$), we have proved that $M \models \varphi$. When φ is a liveness property, the automaton $\mathcal{A}_{\neg \varphi}$ must be an NBA. However, we can specialize the construction of an automaton that represents φ if φ is a *safety property*.

Let $\Sigma = 2^{Prop}$ be a finite alphabet. Given an LTL formula φ over *Prop*, and infinite computation $\pi = \pi_0, \pi_1, \ldots$, recall from Chapter 2.5 that φ is a safety property if:

$$(\forall \pi, \pi \in \Sigma^\omega : \pi \models \varphi \leftrightarrow (\forall i, 0 \leq i : (\exists \beta, \beta \in \Sigma^\omega : \pi_{0\ldots i} \cdot \beta \models \varphi))),$$

where $\cdot$ is the concatenation operator and Σ^ω denotes the set of infinite words (or ω-words) over Σ [145]. So, if φ is violated, there must be some finite prefix $\alpha = \pi_0, \pi_1, \ldots \pi_i$ of the computation π in which this violation occurs. For any infinite computation β over the alphabet Σ, the concatenation of computations $\alpha \cdot \beta$ cannot satisfy φ. In other words, α is a *bad prefix* for $\mathscr{L}(\varphi)$ iff for all $\sigma \in \Sigma^\omega : \alpha \cdot \sigma \notin \mathscr{L}(\varphi)$ [230], where we denote the set of models of φ with $\mathscr{L}(\varphi) = \{w \in \Sigma^\omega \mid w \models \varphi\}$. We denote by $pref(\mathscr{L}(\varphi))$ the set of all such bad prefixes.

While for general properties, the goal of the model checker is to search for bad cycles, for safety properties, the model checker only needs to search for a finite bad prefix. This can be accomplished with a much simpler algorithm, such as a forward or backward reachability check. When φ is a safety property, instead of constructing from φ a Büchi automaton $\mathcal{A}_{\neg\varphi}$ that accepts $\mathscr{L}(\neg\varphi)$, we can construct an automaton on finite words that detects the bad prefixes of φ.

A *Nondeterministic Finite Automaton* (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accepting states. A *Deterministic Finite Automaton* (DFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0, F are defined as for an NFA but $\delta : Q \times \Sigma \rightarrow Q$. A run of a finite automaton $\mathcal{A}$ over a finite computation $\pi = \pi_0, \pi_1, \pi_2, \ldots \pi_n \in \Sigma$ is accepting if it terminates in a state $q_n \in F$.

Theorem 6.1

[144] *Given a safety LTL formula φ, we can construct a deterministic finite automaton*

$\mathcal{A}^d = (Q, \Sigma, \delta, q_0, F)$ such that $|Q|$ is in $2^{2^{O(|\varphi|)}}$, $\Sigma = 2^{Prop}$, and $\mathcal{L}(\mathcal{A}^d)$ is exactly $pref(\mathcal{L}(\varphi))$.

Therefore, when φ is a safety property, we can opt to form an NFA or a DFA corresponding to $\neg\varphi$ instead of an NBA since we only need to construct an automaton that accepts the set of finite prefixes that witness a violation of φ. For example, if $\varphi = \Box good$ then $\neg\varphi = \Diamond\neg good$; intuitively $\neg\varphi$ is witnessed by a finite trace ending in $\neg good$.

LTL → DFA Algorithm   Given a safety formula φ, we form the NBA $\mathcal{A}_\varphi$ using SPOT [140], which was previously shown to be the best available LTL-to-Büchi translator [228]. Similarly to [1] we create via nested depth-first search the state set $empty(\mathcal{A}_\varphi)$ that contains all of the states in $\mathcal{A}_\varphi$ that cannot appear on an accepting run. We use SPOT to remove the set $empty(\mathcal{A}_\varphi)$ from $\mathcal{A}_\varphi$. We then form a finite automaton $\mathcal{A}_\varphi^f$ by re-labeling all remaining states to be accepting. We now have the NFA $\mathcal{A}_\varphi^f$ defined by the quintuple $(Q', \Sigma, \delta', q_0 \cap Q', F \cap Q')$ where $Q' = Q - empty(\mathcal{A}_\varphi)$ and δ' is restricted to $Q' \times \Sigma$.

Theorem 6.2

[225] $\mathcal{A}_\varphi^f$ *rejects precisely the minimal bad prefixes of φ.*

Note that, for model checking, we need an automaton that *accepts* precisely $\mathcal{L}(\neg\varphi)$, which, in this case, corresponds to the set of bad prefixes of φ: $pref(\mathcal{L}(\varphi))$. Customarily, the model checker checks $M \models \varphi$ by first negating φ, then forming the NBA $A_{\neg\varphi}$ since negating an LTL formula avoids the computational complexity of negating its corresponding automaton. However, when φ is a safety formula, we can just as easily perform this negation on-the-fly as we determinize $\mathcal{A}_\varphi^f$ to form our DFA $\mathcal{A}^d$. We do this by adding a single state, q_{stuck} and setting $F = \{q_{stuck}\}$. Then, at each state in $\mathcal{A}_\varphi^f$, whenever there is no legal outgoing transition for some $\sigma \in \Sigma$, we create one to q_{stuck}.

6.3 `Never` Claim Generation

A `never` claim is a Promela code sequence that defines a system behavior that should never happen. A `never` claim essentially corresponds to an automaton. Since we use `never` claims to specify properties that should *never* happen (i.e. bad properties we wish to assert the system does not have), we create a `never` claim corresponding to the negation of the property we wish to hold. In other words, when we create a `never` claim that accepts exactly $\mathscr{L}(\neg\varphi)$ we are stating that it would be a correctness violation of the system if there exists any execution sequence in which $\neg\varphi$ holds. For the system to be considered correct, φ must always hold.

6.3.1 Forming a `Never` claim

To generate a Promela `never` claim for LTL formula φ, Spin translates $\neg\varphi$ into the NBA $\mathcal{A}_{\neg\varphi} = (Q, \Sigma, \delta, q_0, F)$, enumerates the states in Q, labels q_0 with 'init' to designate the state in which the `never` claim starts, labels each $q \in F$ with 'accept,' and implements δ by a nondeterministic choice: for each state, nondeterministically choose from among enabled transitions given the set of propositions true in the current state. Currently, all LTL-to-Promela translators follow this high-level construction. (They vary widely in the details of the formation of $\mathcal{A}_{\neg\varphi}$ as described in Chapter 4.2.1 but output $\mathcal{A}_{\neg\varphi}$ as a Promela `never` claim in the same way.) For example, the `never` claim generated by SPOT for $\varphi = \Box good$ is:

```
1 never { /* <>!good */
2 T0_init:
3   if
4   :: (good) -> goto T0_init
5   :: (!good) -> goto accept_all
6   fi;
7 accept_all:
8   skip
9 }
```

Note that this `never` claim corresponds to an NBA for $\neg\varphi$: a trace violating φ sends the `never` claim into a (single-state) ω-acceptance cycle. Infinite behavior is matched if the claim can reach an ω-acceptance cycle; the run-time flag $-a$ explicitly tells Spin to check for acceptance cycles.

6.3.2 `Never` claims for finite behavior

To prove a system model M satisfies the LTL property $\varphi = (\Box good)$, we create a `never` claim that accepts the negation of this property. Spin can do this automatically using the command `spin -f '![] good'`. Intuitively, the `never` claim generated by this formula would restrict system behavior to those states where ($\Diamond!good$) holds. If any such executions of the system are found, Spin will throw a violation.

In addition to the NBA `never` claims produced by Spin, SPOT, and other tools, `never` claims can be also be used to specify finite automata; the distinction is inherent in the structure of the claim rather than explicitly labeled. Finite behavior is matched if the claim can reach its closing curly brace while executing in lockstep with the system model [188]. Spin automatically checks for this type of `never` claim termination.

A `never` claim corresponding to the NFA that accepts $\neg\varphi$ simply needs to reach its closing curly brace if !*good* is ever true, thus accepting the finite trace corresponding to a correctness violation of the system. Therefore, we could represent this property with a `never` claim as simple [188] as:

```
1  never { /* ![] good == <> ! good */
2          do
3          :: true
4          :: ! good -> break
5          od
6  }
```

Note that we check the above `never` claim using different Spin commands than the

infinite-acceptance version. Specifically, we check for finite acceptance using the following commands:

```
cat Model > pan_in
cat finite_never_claim >> pan_in
spin -a pan_in
gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=128 -DSAFETY -DXUSAFE
    -DNOFAIR -DNXT pan.c
./pan -v -X -m10000 -w19  -A -E -c1
```

In this paper, we construct a Promela `never` claim corresponding to the automaton $\mathcal{A}^d$. We now describe novel alternatives for constructing $\mathcal{A}^d$ when φ is a safety property.

6.3.3 Determinization

We experiment with forming our DFA $\mathcal{A}^d$ in two ways [1]. Firstly, we can explicitly determinize our NFA using the subset construction via the `BRICS Automaton` tool [231], which we call a `det` construction. Secondly, we can create a Promela `never` claim from our nondeterministic automaton with added code to essentially perform the subset construction on-the-fly (`nondet` construction). To do this, we construct a run $P_0, P_1, \ldots, P_n$ of $\mathcal{A}^d$ such that $P_0 = \{q_0\}$ and $P_{i+1} = \bigcup_{q \in P_i} \delta(q, \sigma_i)$. Studying the trade-off between these two constructions is one of the contributions of this paper.

6.3.4 State minimization

We can opt to utilize a tool that constructs a minimal equivalent $\mathcal{A}^d$ such as the `BRICS Automaton` tool [231], which we call a `min` construction. The extra steps of determinization and minimization may incur a nontrivial computational cost during the construction of $\mathcal{A}^d$. A key question is whether the cost of minimization and the associated reduction of time required by the model-checking step lead to a better amortized verification cost if we can create $\mathcal{A}^d$ once and use it for model checking many times.

6.3.5   Alphabet representation [1]

SPOT labels the transitions of $\mathcal{A}^d$ with Boolean formulas over the alphabet of $\mathcal{A}^d$, $\Sigma =$ 2^{Prop}. We cannot use Boolean formulas for transition labels in all of our constructions since automata-theoretic algorithms for determinization and minimization of NBAs require comparing elements of Σ.[2] Therefore, we adapt the methods of [1] for describing the alphabet of $\mathcal{A}^d$ in terms of 16-bit integers.

BDD-based representation

We can represent the Boolean formulas comprising the alphabet utilizing *Ordered Binary Decision Diagrams (OBDDs).* Recall from Chapter 2.9 that an OBDD is a rooted, directed, acyclic graph with internal variable-node vertices of out-degree two, labeled by some sufficient subset of the *n* variables of the corresponding Boolean formula, and exactly two terminal vertices, 0 and 1. A root-to-leaf path through the OBDD follows a total order of the variables, representing a valuation of the Boolean formula.

   We implement this approach as follows. First, we obtain references to all Boolean formulas that appear as transition labels in $\mathcal{A}^f$ using SPOT's `spot::tgba_reachable` `_iterator_breadth_first::process_link()` function. Second, we assign a unique integer label to the OBDD representation of each Boolean formula (up to $2^{|\varphi|}$ in the worst case) with the help of SPOT's `spot::tgba_succ_iterator::current_condition()` function and the canonicity property of OBDDs for a given variable order. We can then define $\mathcal{A}^f$ using integers to form the alphabet for our BDD-based representation: $\mathcal{A}^f =$ $(Q, \{0, \dots, 2^{|\varphi|}\}, \delta_{bdd}, q_0, F)$ where $q' \in \delta_{bdd}(q, z)$ iff $q' \in \delta(q, \sigma)$ and the integer z labels the Boolean formula σ.

---

[2]`BRICS Automaton` represents the alphabet of the automaton as Unicode characters, which have 1-to-1 correspondence to the set of 16-bit integers.

Assignment-based representation

Traditionally, we represent Boolean formulas in terms of their satisfying truth *assignments*.
An *assignment* to the set of propositions in *Prop* is an *n*-bit vector $\mathbf{a} = [a_1, a_2, \ldots, a_n]$; $\mathbf{a}$
satisfies Boolean formula φ over $\{p_1, p_2, \ldots, p_n\} \in Prop$ iff assigning the values $(a_1, a_2, \ldots, a_n)$
to the propositions in *Prop* causes φ to evaluate to *true*. We can define a function I map-
ping from the set of such possible assignments to the set of integers by evaluating $\mathbf{a}$ as the
integer whose binary representation is $\mathbf{a}$. Thus, $I(\mathbf{a}) = a_1 2^{n-1} + a_2 2^{n-2} + \ldots + a_n 2^0$. Let
$sat(\varphi) = \{I(\mathbf{a}) : \mathbf{a} \text{ satisfies } \varphi\}$. We can then define $\mathcal{A}^f$ utilizing assignment-based repre-
sentation: $\mathcal{A}^f_{abr} = (Q, \{0, \ldots, 2^n - 1\}, \delta_{abr}, q_0, F)$ where $q' \in \delta_{abr}(q, z)$ iff $q' \in \delta(q, \sigma)$ and
$z \in sat(\sigma)$ for some integer z and Boolean formula σ.

Forming $\mathcal{A}^d$

We use `BRICS Automaton` to form $\mathcal{A}^d_{abr}$ and $\mathcal{A}^d_{bdd}$ from $\mathcal{A}^f_{abr}$ and $\mathcal{A}^f_{bdd}$, respectively.
We then convert the automata alphabets from integers corresponding to their alphabet-
based or bdd-based representations back to Boolean formulas that we can use in forming
transition labels as we produce a corresponding Promela `never` claim. In cases when $\mathcal{A}^d$
is a minimized determinized automaton formed from $\mathcal{A}^f_{abr}$ we may optionally employ edge
abbreviation before forming the `never` claim.

Edge Abbreviation

When we create `never` claims using the assignment-based representation, we can create
redundant transitions; a state can easily have $2^{|\Sigma|}$ transitions, one for every valuation of
the set of atomic propositions. We can employ edge abbreviation by creating a disjunc-
tion of all of the transition labels of transitions that share the same source and destination
states, removing all of these transitions from $\mathcal{A}^d$, and replacing each set of redundant tran-

sitions with a single transition labeled by the disjunction of their labels. For each such disjunction, we utilize SPOT's built-in `formula_to_bdd()` function to create a BDD representing the disjunction, extract a simplified formula from the BDD via the reverse `bdd_to_formula()` function, and then label the associated transition by this new, usually simpler, formula. In effect, we replace the original set of transitions with an abbreviated set of transitions.

In the case that a state in a `never` claim formed using the assignment-based representation has $t < 2^{|\Sigma|}$ transitions, the remaining $2^{|\Sigma|} - t$ valuations of the set of atomic propositions cause $\mathcal{A}^d$ to transition to the accepting state. In Promela, this is encapsulated with an `else` transition to the accepting state; an `else` transitions is enabled if and only if no other transitions are enabled in the same state. However, we can also create a label for this transition from a state q by taking the negation of the disjunction of all of the abbreviated transitions whose source is q and using SPOT's built-in BDD functions to reduce this formula. If the resulting formula is `false`, then there is no transition from q to the accepting state and we can delete the `else` transition; if not, the resulting formula is the label of this transition. Explicitly labeling these transitions to accepting states allows us to prune any *trap states* from $\mathcal{A}^d$ where a trap state q is defined to be a non-accepting, non-initial state with exactly one self-looping transition from q back to q. In the semantics of Promela, once we have eliminated all `else` transitions, we can simply delete all trap states and any transitions to or from them, resulting in an equivalent `never` claim representing $\mathcal{A}^d$ with a smaller code signature. Note that this construction retains the deterministic nature of $\mathcal{A}^d$; instead of transitioning to a trap state with a non-accepting loop, the automaton will simply fail immediately whenever Spin detects that no transitions are enabled.

6.3.6 `Never` claim encodings

We introduce 26 ways of encoding automata as Promela `never` claims. We form these encodings by combining our `never` claim adaptations of the constructions for transition direction (`front` vs `back`), determinism (`det` vs `nondet`), state minimization (`min` vs `nomin`), and alphabet representation (`bdd` vs `abr`) from [1] with the options to encode `never` claim states either using Promela state labels or integer state numbers (`state` vs `number`), to employ either finite or infinite acceptance conditions (`fin` vs `inf`), and to reduce the size of the generated `never` claim via edge abbreviation and trap-state elimination (`ea`). We illustrate our encodings below for a benchmark safety formula we name ButtonPushX (Chapter 3.6.1).

Nondeterministic encodings

We introduce 12 novel encodings that encode $\mathcal{A}^f$ as a `never` claim while performing determinization (i.e. creating $\mathcal{A}^d$) on-the-fly. The major difference between `nondet` and `det never` claims is that `nondet never` claims maintain an array used for tracking all possible runs of the automaton given the valuations of the state variables and report failure when there are no possible runs. This tracking array is not needed in `det never` claims since they have already been determinized and for any computation, there is only one possible run. We must encode `nondet never` claims using state numbers, similarly to [1]. Using state numbers enables us to track the possible current and next states; state labels and edge abbreviation algorithms are not compatible with this construction. We can encode the transition relations either in either a `front` fashion, where for any state q we enumerate the outgoing transitions from q, or in a `back` fashion, where for any state q we reason over the incoming transitions that could have led us to q.

The `front_nondet` encoding uses an `if` statement to check each outgoing transition

from each possible current state and marks all possible next states in the next_state array. If there are no possible next states, the automaton fails. For never claims with finite acceptance conditions, this is accomplished by breaking from the do loop and coming to the end } of the claim, as shown in Listing 6.1.

```
1   /*LTL formula: (!(X ((p0 & p1) R p2)))*/
2   int i = 0;
3   bool not_stuck = false;
4
5   /*Declare state arrays; they are automatically initialized to 0*/
6   bool current_state [3];
7   bool next_state [3];
8   never {
9     /*This next line happens in time -1; one step before the first
10       step of the system model*/
11    next_state[2] = 1; /*initialize current to the initial state*/
12
13    do
14      :: atomic{
15        /*First, swap of current_state and next_state*/
16        i = 0;
17        do
18          :: (i < 3 ) ->
19              current_state[i] = next_state[i];
20              i++;
21          :: (i >= 3) -> break;
22        od;
23        /*reset next_state*/
24        i = 0;
25        do
26          :: (i < 3) ->
27              next_state[i] = 0;
28              i++;
29          :: (i >= 3) -> break;
30        od;
31        /*Second, fill in next_state array*/
32        if
33          :: current_state[2] ->
34              if
35                :: (1  )
36                    -> next_state[1] = 1;
37                :: else -> skip;
```

```
38            fi;
39          :: else -> skip;
40       fi;
41       if
42          :: current_state[0] ->
43             if
44                :: (1  )
45                   -> next_state[0] = 1;
46                :: else -> skip;
47             fi;
48          :: else -> skip;
49       fi;
50       if
51          :: current_state[1] ->
52             if
53                :: (p0 && p1 && p2  )
54                   -> next_state[0] = 1;
55                :: else -> skip;
56             fi;
57             if
58                :: ((p2 && !p0) || (p2 && !p1)  )
59                   -> next_state[1] = 1;
60                :: else -> skip;
61             fi;
62          :: else -> skip;
63       fi;
64       /*Third, check if we're stuck*/
65       i = 0;
66       not_stuck = false;
67       do
68          :: (i < 3) ->
69             not_stuck = not_stuck || next_state[i];
70             i++;
71          :: (i >= 3) -> break;
72       od;
73       if
74          :: (! not_stuck) -> break;
75          :: else -> skip;
76       fi;
77    }
78  od;
79 }
```

Listing 6.1: Illustrating front_nondet_nomin_bdd_number_fin never claim encoding of the ButtonPushX formula. Note this encoding utilizes finite acceptance.

The `back_nondet` encoding similarly begins by initializing the arrays tracking the possible current and next states except now a next state is enabled by checking that there is an enabled incoming transition from an enabled current state and a valid transition label. If there are no possible next states, the automaton fails. For `never` claims with infinite acceptance conditions, we loop infinitely often if the property holds. This is accomplished by transitioning to an accepting state with a self-loop, as shown in Listing 6.2.

```
1  /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2  int i = 0;
3
4  /*Declare state arrays
5    They are automatically initialized to 0*/
6  bool current_state [3];
7  bool next_state [3];
8  never {
9
10 S0_init: /*initialize current here*/
11 atomic {
12   current_state[0] = 1;
13   next_state[0] = 0;
14   next_state[1] = ( current_state[2] && (p0 && p1 && p2)) ||
15                   ( current_state[1] && (1));
16   next_state[2] = ( current_state[0] && (1)) ||
17                   ( current_state[2] && ((p2&&!p0)||(p2&&!p1)));
18
19   /* if any next state is enabled, loop */
20   /* Note that this if-statement will choose nondeterministically
21      from among the true guards, but that's OK since multiple
22      guards go to the same place*/
23   if
24     :: next_state[0] -> goto S1;
25     :: next_state[1] -> goto S1;
26     :: next_state[2] -> goto S1;
27     :: else -> goto accept_all;
28   fi;
29   }
30
31 S1: /*loop here forever if property holds*/
32 atomic {
33   /*update: current_state = next_state*/
34   i = 0;
```

```
35   do
36      :: (i < 3) ->
37          current_state[i] = next_state[i];
38          i++;
39      :: (i >= 3) -> break;
40   od;
41
42   next_state[0] = 0;
43   next_state[1] = ( current_state[2] && (p0 && p1 && p2)) ||
44                   ( current_state[1] && (1));
45   next_state[2] = ( current_state[0] && (1)) ||
46                   ( current_state[2] && ((p2&&!p0)||(p2&&!p1)));
47
48   /* if any next state is enabled, loop */
49   if
50     :: next_state[0] -> goto S1;
51     :: next_state[1] -> goto S1;
52     :: next_state[2] -> goto S1;
53     :: else -> goto accept_all;
54   fi;
55     }
56
57 accept_all: /*signal property violation by omega-looping here*/
58     skip;
59 }
```

Listing 6.2: Illustrating `back_nondet_min_bdd_number_inf` encoding of the Button-PushX formula. Note this encoding utilizes infinite acceptance.

Deterministic encodings

We introduce 14 novel deterministic encodings that presume $\mathcal{A}^d$ has been minimized and determinized using assignment-based encoding. Since the `det` encodings do not need to determinize $\mathcal{A}^d$ on the fly, we have two options for encoding the states. We can utilize numbers (`number`) to refer to the states implicitly as we do for the `nondet` encodings, except that since there is only one possible run of the automaton we need only one integer each to refer to the current and next states rather than two arrays of integers as we used for the `nondet` encodings. Alternatively, we can use Spin's standard state-label format

coupled with `goto` statements to transition between states. We illustrate each of these two state representations in turn.

The `back_det` encoding uses state numbers. The `never` claim begins by calculating the `system_state_index`, the integer corresponding to the current valuation of the system variables. Like its `back_nondet` counterpart, it transitions by checking for an enabled incoming transition from the enabled current state, except in the `back_det` `never` claim the current and next states are represented as single integers instead of arrays. For `never` claims with finite acceptance conditions, acceptance is accomplished by breaking from the `do` loop and coming to the end } of the claim, as shown in Listing 6.3.

```
1  /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2  int current_state = 2;
3  int next_state = 2;
4  int system_state_index = 0;
5  never {
6    next_state = 2; /*initialize current to the initial state here*/
7
8    do
9      :: atomic {
10       current_state = next_state; /*update state*/
11       next_state = -1; /*reset*/
12
13       /*Calculate the system state index*/
14       system_state_index = 0; /*reset*/
15       system_state_index=system_state_index+ ((p0) -> (1 << 2):0);
16       system_state_index=system_state_index+ ((p1) -> (1 << 1):0);
17       system_state_index=system_state_index+ ((p2) -> (1 << 0):0);
18       if
19         :: (((current_state == 2) && (system_state_index == 5)) ||
20             ((current_state == 2) && (system_state_index == 7)) ||
21             ((current_state == 0) && (system_state_index == 1)) ||
22             ((current_state == 0) && (system_state_index == 5)) ||
23             ((current_state == 2) && (system_state_index == 6)) ||
24             ((current_state == 2) && (system_state_index == 0)) ||
25             ((current_state == 2) && (system_state_index == 3)) ||
26             ((current_state == 2) && (system_state_index == 4)) ||
27             ((current_state == 0) && (system_state_index == 3)) ||
28             ((current_state == 2) && (system_state_index == 1)) ||
29             ((current_state == 2) && (system_state_index == 2)))
```

```
30                    -> next_state = 0;
31           :: (((current_state == 0) && (system_state_index == 7)) ||
32               ((current_state == 1) && (system_state_index == 0)) ||
33               ((current_state == 1) && (system_state_index == 1)) ||
34               ((current_state == 1) && (system_state_index == 2)) ||
35               ((current_state == 1) && (system_state_index == 3)) ||
36               ((current_state == 1) && (system_state_index == 4)) ||
37               ((current_state == 1) && (system_state_index == 5)) ||
38               ((current_state == 1) && (system_state_index == 6)) ||
39               ((current_state == 1) && (system_state_index == 7)))
40               -> next_state = 1;
41          :: else break;
42       fi;
43     }
44   od;
45 }
```

Listing 6.3: Illustrating `back_det_min_abr_number_fin` encoding of the Button-PushX formula.

The `front_det_switch_number_fin` encoding uses a series of `if` statements, the closest Promela construction to a C-like `switch` statement, to check for enabled outgoing transitions from the current state. As in Listing 6.3, this encoding uses state numbers as identifiers and employs the Promela finite acceptance condition where the claim is accepted if the `never` claim reaches its closing curly brace, as shown in Listing 6.4. Note that this encoding is the closest Promela equivalent to the winning SystemC encoding for LTL monitors [1].

```
1 /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2 int current_state = 2;
3 int next_state = 2;
4 int system_state_index = 0;
5 never {
6   next_state = 2; /*initialize current to the initial state here*/
7
8   do
9     :: atomic {
10        current_state = next_state; /*update state*/
11        next_state = -1; /*reset*/
12
13        /*Calculate the system state index*/
```

```
14          system_state_index = 0; /*reset*/
15          system_state_index=system_state_index+((p0) -> (1 << 2):0);
16          system_state_index=system_state_index+((p1) -> (1 << 1):0);
17          system_state_index=system_state_index+((p2) -> (1 << 0):0);
18          if
19            :: (current_state == 2) ->
20                if
21                  :: (system_state_index == 5 )
22                     -> next_state =   0;
23                  :: (system_state_index == 7 )
24                     -> next_state =   0;
25                  :: (system_state_index == 6 )
26                     -> next_state =   0;
27                  :: (system_state_index == 0 )
28                     -> next_state =   0;
29                  :: (system_state_index == 3 )
30                     -> next_state =   0;
31                  :: (system_state_index == 4 )
32                     -> next_state =   0;
33                  :: (system_state_index == 1 )
34                     -> next_state =   0;
35                  :: (system_state_index == 2 )
36                     -> next_state =   0;
37                  :: else break;
38                fi;
39            :: (current_state == 0) ->
40                if
41                  :: (system_state_index == 7 )
42                     -> next_state =   1;
43                  :: (system_state_index == 1 )
44                     -> next_state =   0;
45                  :: (system_state_index == 5 )
46                     -> next_state =   0;
47                  :: (system_state_index == 3 )
48                     -> next_state =   0;
49                  :: else break;
50                fi;
51            :: (current_state == 1) ->
52                if
53                  :: (system_state_index == 0 )
54                     -> next_state =   1;
55                  :: (system_state_index == 1 )
56                     -> next_state =   1;
57                  :: (system_state_index == 2 )
58                     -> next_state =   1;
```

```
59                       :: (system_state_index == 3 )
60                          -> next_state =   1;
61                       :: (system_state_index == 4 )
62                          -> next_state =   1;
63                       :: (system_state_index == 5 )
64                          -> next_state =   1;
65                       :: (system_state_index == 6 )
66                          -> next_state =   1;
67                       :: (system_state_index == 7 )
68                          -> next_state =   1;
69                       :: else break;
70                  fi;
71           fi;
72           }
73    od;
74 }
```

Listing 6.4: Illustrating front_det_switch_number_fin never claim encoding of the ButtonPushX formula.

The front_det_switch_number_inf encoding modifies the encoding in Listing 6.4 to employ infinite acceptance. If there are no possible next states, the automaton fails by looping at the accept_stuck label, as shown in Listing 6.5.

```
1 /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2 int current_state = 2;
3 int next_state = 2;
4 int system_state_index = 0;
5 never {
6
7 S0_init:/*initialize current here*/
8   atomic {
9   current_state = 2;
10
11   /*Calculate the system state index*/
12   system_state_index = 0; /*reset*/
13   system_state_index = system_state_index + ((p0) -> (1 << 2):0);
14   system_state_index = system_state_index + ((p1) -> (1 << 1):0);
15   system_state_index = system_state_index + ((p2) -> (1 << 0):0);
16   if
17       :: (system_state_index == 5 )
18         -> next_state =    0; goto S1;
19       :: (system_state_index == 7 )
20         -> next_state =    0; goto S1;
```

```
21        :: (system_state_index == 6 )
22          -> next_state =    0; goto S1;
23        :: (system_state_index == 0 )
24          -> next_state =    0; goto S1;
25        :: (system_state_index == 3 )
26          -> next_state =    0; goto S1;
27        :: (system_state_index == 4 )
28          -> next_state =    0; goto S1;
29        :: (system_state_index == 1 )
30          -> next_state =    0; goto S1;
31        :: (system_state_index == 2 )
32          -> next_state =    0; goto S1;
33        :: else
34          -> goto accept_stuck;
35      fi;
36      }
37
38  S1: /*loop here forever if property holds*/
39      atomic {
40      current_state = next_state; /*update state*/
41
42      /*Calculate the system state index*/
43      system_state_index = 0; /*reset*/
44      system_state_index = system_state_index + ((p0) -> (1 << 2):0);
45      system_state_index = system_state_index + ((p1) -> (1 << 1):0);
46      system_state_index = system_state_index + ((p2) -> (1 << 0):0);
47      if
48        :: (current_state == 2) ->
49          if
50          :: (system_state_index == 5 )
51            -> next_state =    0; goto S1;
52          :: (system_state_index == 7 )
53            -> next_state =    0; goto S1;
54          :: (system_state_index == 6 )
55            -> next_state =    0; goto S1;
56          :: (system_state_index == 0 )
57            -> next_state =    0; goto S1;
58          :: (system_state_index == 3 )
59            -> next_state =    0; goto S1;
60          :: (system_state_index == 4 )
61            -> next_state =    0; goto S1;
62          :: (system_state_index == 1 )
63            -> next_state =    0; goto S1;
64          :: (system_state_index == 2 )
65            -> next_state =    0; goto S1;
```

```
66        :: else
67          -> goto accept_stuck;
68        fi;
69      :: (current_state == 0) ->
70        if
71        :: (system_state_index == 7 )
72          -> next_state =     1; goto S1;
73        :: (system_state_index == 1 )
74          -> next_state =     0; goto S1;
75        :: (system_state_index == 5 )
76          -> next_state =     0; goto S1;
77        :: (system_state_index == 3 )
78          -> next_state =     0; goto S1;
79        :: else
80          -> goto accept_stuck;
81        fi;
82      :: (current_state == 1) ->
83        if
84        :: (system_state_index == 0 )
85          -> next_state =     1; goto S1;
86        :: (system_state_index == 1 )
87          -> next_state =     1; goto S1;
88        :: (system_state_index == 2 )
89          -> next_state =     1; goto S1;
90        :: (system_state_index == 3 )
91          -> next_state =     1; goto S1;
92        :: (system_state_index == 4 )
93          -> next_state =     1; goto S1;
94        :: (system_state_index == 5 )
95          -> next_state =     1; goto S1;
96        :: (system_state_index == 6 )
97          -> next_state =     1; goto S1;
98        :: (system_state_index == 7 )
99          -> next_state =     1; goto S1;
100       :: else
101         -> goto accept_stuck;
102       fi;
103   fi;
104   }
105 accept_stuck: /*signal property violation by omega-looping here*/
106   skip;
107 }
```

Listing 6.5: Illustrating front_det_switch_number_inf never claim encoding of the ButtonPushX formula. It employs the Promela acceptance-cycle acceptance condition.

Alternatively, we can encode an equivalent `never` claim to that in Listing 6.5 without using any state numbers, by taking advantage of Promela's constructs for representing automata states. The `front_det_switch_state_inf` encoding in Listing 6.6 transitions to program labels corresponding to the names of the states in $\mathcal{A}^d$. The initial state is labeled "`init`" and appears first, the accepting state is labeled "`accept`," and all other states are assigned unique names. The `front_det_switch_state_inf` encoding also relies on Spin's implementation of infinite acceptance where the model checker searches for infinite acceptance cycles that will only be found by transitioning to the `accept_stuck` state and then looping there forever.

```
1  /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2  int system_state_index = 0;
3  never {
4
5  init_S2:
6    atomic {
7      system_state_index = 0; /*reset*/
8      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
9      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
10     system_state_index= system_state_index + ((p2) -> (1 << 0):0);
11     if
12       :: (system_state_index == 5 )
13         -> goto   S0;
14       :: (system_state_index == 7 )
15         -> goto   S0;
16       :: (system_state_index == 6 )
17         -> goto   S0;
18       :: (system_state_index == 0 )
19         -> goto   S0;
20       :: (system_state_index == 3 )
21         -> goto   S0;
22       :: (system_state_index == 4 )
23         -> goto   S0;
24       :: (system_state_index == 1 )
25         -> goto   S0;
26       :: (system_state_index == 2 )
27         -> goto   S0;
28       :: else
29         -> goto accept_stuck;
```

```
30      fi;
31    }
32  S0:
33    atomic {
34      system_state_index = 0; /*reset*/
35      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
36      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
37      system_state_index= system_state_index + ((p2) -> (1 << 0):0);
38      if
39        :: (system_state_index == 7 )
40          -> goto    S1;
41        :: (system_state_index == 1 )
42          -> goto    S0;
43        :: (system_state_index == 5 )
44          -> goto    S0;
45        :: (system_state_index == 3 )
46          -> goto    S0;
47        :: else
48          -> goto accept_stuck;
49      fi;
50    }
51  S1:
52    atomic {
53      system_state_index = 0; /*reset*/
54      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
55      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
56      system_state_index= system_state_index + ((p2) -> (1 << 0):0);
57      if
58        :: (system_state_index == 0 )
59          -> goto    S1;
60        :: (system_state_index == 1 )
61          -> goto    S1;
62        :: (system_state_index == 2 )
63          -> goto    S1;
64        :: (system_state_index == 3 )
65          -> goto    S1;
66        :: (system_state_index == 4 )
67          -> goto    S1;
68        :: (system_state_index == 5 )
69          -> goto    S1;
70        :: (system_state_index == 6 )
71          -> goto    S1;
72        :: (system_state_index == 7 )
73          -> goto    S1;
74        :: else
```

```
75            -> goto accept_stuck;
76       fi;
77     }
78 accept_stuck: /*signal property violation by omega-looping here*/
79     skip;
80 }
```

Listing 6.6: Illustrating front_det_switch_min_abr_state_inf never claim encoding of the ButtonPushX formula. This version employs the Promela notion of states and the Promela acceptance-cycle acceptance condition.

The front_det_switch_state_fin encoding changes the equivalent encoding in Listing 6.6 only in that it employs the Promela finite acceptance condition where the claim is accepted if the never claim reaches its closing curly brace. Note that we can eliminate the accept_stuck state and instead accept based on never claim termination at the done label, as shown in Listing 6.7.

```
1  /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2  int system_state_index = 0;
3  never {
4
5  init_S2:
6    atomic {
7      system_state_index = 0; /*reset*/
8      system_state_index= system_state_index + ((p0) -> (1 << 2):0);
9      system_state_index= system_state_index + ((p1) -> (1 << 1):0);
10     system_state_index= system_state_index + ((p2) -> (1 << 0):0);
11     if
12       :: (system_state_index == 5 )
13         -> goto   S0;
14       :: (system_state_index == 7 )
15         -> goto   S0;
16       :: (system_state_index == 6 )
17         -> goto   S0;
18       :: (system_state_index == 0 )
19         -> goto   S0;
20       :: (system_state_index == 3 )
21         -> goto   S0;
22       :: (system_state_index == 4 )
23         -> goto   S0;
24       :: (system_state_index == 2 )
25         -> goto   S0;
```

```
26      :: (system_state_index == 1 )
27         -> goto   S0;
28      :: else -> goto done;
29    fi;
30   }
31 S0:
32   atomic {
33     system_state_index = 0; /*reset*/
34     system_state_index= system_state_index + ((p0) -> (1 << 2):0);
35     system_state_index= system_state_index + ((p1) -> (1 << 1):0);
36     system_state_index= system_state_index + ((p2) -> (1 << 0):0);
37     if
38       :: (system_state_index == 7 )
39         -> goto   S1;
40       :: (system_state_index == 1 )
41         -> goto   S0;
42       :: (system_state_index == 5 )
43         -> goto   S0;
44       :: (system_state_index == 3 )
45         -> goto   S0;
46       :: else -> goto done;
47     fi;
48   }
49 S1:
50   atomic {
51     system_state_index = 0; /*reset*/
52     system_state_index= system_state_index + ((p0) -> (1 << 2):0);
53     system_state_index= system_state_index + ((p1) -> (1 << 1):0);
54     system_state_index= system_state_index + ((p2) -> (1 << 0):0);
55     if
56       :: (system_state_index == 0 )
57         -> goto   S1;
58       :: (system_state_index == 1 )
59         -> goto   S1;
60       :: (system_state_index == 2 )
61         -> goto   S1;
62       :: (system_state_index == 3 )
63         -> goto   S1;
64       :: (system_state_index == 4 )
65         -> goto   S1;
66       :: (system_state_index == 5 )
67         -> goto   S1;
68       :: (system_state_index == 6 )
69         -> goto   S1;
70       :: (system_state_index == 7 )
```

```
71            -> goto   S1;
72          :: else -> goto done;
73        fi;
74     }
75  done: /*signal property violation by landing here*/
76      skip;
77  }
```

Listing 6.7: Illustrating `front_det_switch_min_abr_state_fin` never claim encoding of the ButtonPushX formula.

Reducing Deterministic Encoding Size

The encodings above have as many as $2^{|\Sigma|}$ transitions per state, sometimes fewer if multiple valuations of Σ lead to automaton acceptance. To study the effect of code size on model-checking performance of `never` claims, for `det` encodings utilizing state labels we can create equivalent `never` claims with more compact code by abbreviating the transitions. Employing edge abbreviation (`ea`) to the `front_det_switch_state_fin` encoding in Listing 6.7 gives us the `never` claim in Listing 6.8.

```
1  /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2  never {
3
4  init_S2:
5    atomic {
6      if
7        :: (1 )
8          -> goto   S0;
9        :: else -> goto done;
10       fi;
11    }
12  S0:
13    atomic {
14      if
15        :: ((p2 && !p0) || (!p1 && p2) )
16          -> goto   S0;
17        :: (p0 && p1 && p2 )
18          -> goto   S1;
19        :: else -> goto done;
20       fi;
```

```
21   }
22 S1:
23   atomic {
24     if
25       :: (1 )
26         -> goto   S1;
27       :: else -> goto done;
28     fi;
29   }
30 done: /*signal property violation by landing here*/
31   skip;
32 }
```

Listing   6.8:   Illustrating   an   intermediate   version   of   the
`front_det_switch_min_abr_ea_state_fin` never claim encoding of the
ButtonPushX formula, leaving off the trap state elimination algorithm.

We take advantage of the Promela semantics property that transitioning to a terminal

error state and failing to find such a transition are equivalent. This enables us to further re-

duce the code size for finite-acceptance `never` claims by employing trap state elimination

as we are abbreviating the edges to produce the final, edge abbreviated, `never` claim in

Listing 6.9.

```
 1 /*LTL formula: (!(X ((p0 & p1) R p2)))*/
 2 never {
 3 init_S2:
 4   atomic {
 5     if
 6       :: (1) -> goto S0;
 7     fi;
 8   }
 9 S0:
10   atomic {
11     if
12       :: (!p2) -> goto done;
13       :: ((!p0 && p2) || (!p1 && p2)) -> goto S0;
14     fi;
15   }
16 done: /*signal property violation by landing here*/
17   skip;
```

```
18  }
```

Listing 6.9: Illustrating the final `front_det_switch_min_abr_ea_state_fin` `never` claim encoding of the ButtonPushX formula.

In the encodings discussed above, the transition function of the automaton is encoded using `if` statements. Alternatively, we can format our `front` encodings by declaring a state look-up table in system memory that allows us to find the next state by indexing the current state and an integer in $\{0, \ldots, 2^n - 1\}$ called the `system_state_index` corresponding to the current valuation of the set of system variables. We effectively streamline the code of the `never` claim following the table declaration upon initialization; we can look up the next state in the table in one operation. The idea behind this encoding is to potentially avoid overhead associated with large `if` statements. The `front_det_memory` `_table` encoding declares the state-transition look-up table directly as a one-dimensional, row-major array, illustrated in Listing 6.10.

```
1   /* LTL formula: (!(X ((p0 & p1) R p2)))*/
2   int current_state = 0;
3   int next_state = 0;
4   int system_state_index = 0;
5   int table[24];
6   never {
7
8   S0_init:/*initialize current here*/
9     atomic {
10    table[0]  = 2;   table[1]  = 2;   table[2]  = 2;   table[3]  = 2;
11    table[4]  = 2;   table[5]  = 2;   table[6]  = 2;   table[7]  = 2;
12    table[8]  = 1;   table[9]  = 1;   table[10] = 1;   table[11] = 1;
13    table[12] = 1;   table[13] = 1;   table[14] = 1;   table[15] = 1;
14    table[16] = -1;  table[17] = 2;   table[18] = -1;  table[19] = 2;
15    table[20] = -1;  table[21] = 2;   table[22] = -1;  table[23] = 1;
16
17    /*Calculate the system state index*/
18    system_state_index = 0; /*reset*/
19    system_state_index = system_state_index + ((p0) -> (1 << 2):0);
20    system_state_index = system_state_index + ((p1) -> (1 << 1):0);
21    system_state_index = system_state_index + ((p2) -> (1 << 0):0);
22
```

```
23    /*Lookup the next state in the table*/
24    next_state = table[current_state * 8 + system_state_index];
25    if
26      :: (next_state == -1) -> goto accept_stuck;
27      :: else -> goto S1;
28    fi;
29    }
30
31 S1: /*loop here forever if property holds*/
32    atomic {
33    current_state = next_state; /*update state*/
34    next_state = -1; /*reset*/
35
36    /*Calculate the system state index*/
37    system_state_index = 0; /*reset*/
38    system_state_index = system_state_index + ((p0) -> (1 << 2):0);
39    system_state_index = system_state_index + ((p1) -> (1 << 1):0);
40    system_state_index = system_state_index + ((p2) -> (1 << 0):0);
41
42    /*Lookup the next state in the table*/
43    next_state = table[current_state * 8 + system_state_index];
44        if
45          :: (next_state == -1) -> goto accept_stuck;
46          :: else -> goto S1;
47        fi;
48    }
49
50 accept_stuck: /*signal property violation by omega-looping here*/
51    skip;
52 }
```

Listing 6.10: Illustrating front_det_memory_table_min_abr_inf encoding of the ButtonPushX formula.

6.3.7 Configuration space

The different options allow 26 possible combinations for generating a never claim, summarized in Table 6.1.

State Minimization	Alphabet Representation	Automaton Acceptance	Never Claim Encoding	State Representation
no	BDDs	finite / infinite	`front_nondet` `back_nondet`	number
yes	assignments		`front_nondet` `back_nondet` `back_det` `front_det_memory_table`	
			`front_det_switch`	state number
	assignments with edge abbreviation		`back_det` `front_det_memory_table`	number

Table 6.1 : The configuration space for generating `never` claims.

6.4   Experimental Methods

Platform   We ran all tests on the Shared University Grid at Rice (SUG@R), an Intel Xeon compute cluster.[3] SUG@R is comprised of 134 SunFire x4150 nodes, each with two quad-core Intel Xeon processors running at 2.83GHz and 16GB of RAM per processor. The OS is Red Hat Enterprise 5 Linux, 2.6.18 kernel. Each test was run with exclusive access to one node. Times were measured using the Unix `time` command.

Benchmarks   We define two major categories of model-checking benchmarks employing the linearly-sized Universal Model (UM) from Chapter 3.5.1: one in which we scale the model and one in which we scale the specifications. In the former case, we check all encodings against the set of 14 model-scaling benchmarks described in Chapter 3.6.1 with scaled universal models. In the latter case, we check a universal model with 30 variables and 1,073,741,824 states against the scalable formulas described in Chapter 3.6.2.

---

[3]`http://rcsg.rice.edu/sugar/`

It is interesting to note that we cannot use the counter formulas from Chapter 3.2 in the set of formula-scaling benchmarks. Recall from Chapter 4.3.3 that we do not negate the counter formulas before creating `never` claims from them because we want Spin to check for the single counterexample satisfying the input counter formula, which is a challenging task. It is precisely this aspect of the counter formulas that make them excellent benchmarks for LTL satisfiability checking yet poor benchmarks for LTL model checking of safety formulas. Recall the basic process for model checking of safety formulas: we take a safety property φ that states "something bad never happens," we negate phi (now we have "something bad happens"), we create a `never` claim from $\neg\varphi$, and model checking with Spin returns a counterexample that is a finite trace from a start state to a state in which something bad happens. In this process, our counter formulas correspond to $\neg\varphi$; to use them as benchmarks for model checking instead of satisfiability checking, we would have to negate them first, to get our desired input formula φ. While it is evident that the counter formulas, in their positive form, are safety properties (indeed, two of them are syntactically safe), the negations of our counter formulas are *not* safety properties. (Again, this can be easily confirmed via the SPOT command `ltl2tgba -O`.) Incidentally, we also cannot use the positive counter formulas as φ instead. Negated counter formulas make poor model checking benchmarks in general; intuitively it is very easy to find a counterexample trace that is *not* a binary counter.

Test Method   We encode every benchmark LTL formula as a set of Promela `never` claims using SPOT and the set of our novel encodings. We also experimented with scheck [189] encodings; that tool produced too many bugs to be included in the data presented here. Each `never` claim, was model checked by the Spin back-end.[4]

---

[4]We also investigated using the SPOT back-end; SPOT is unable to analyze Promela `never` claims at the time of this writing.

We measure model checking time separately from the times for various compilation stages. This is important for two reasons. It is relevant for regression testing and system debugging applications where the system is repeatedly changed but model checked against the same specifications. It is also essential for demonstrating our claim that deterministic encoding of LTL safety formulas can reduce model checking time; it is clear that we are not, for example, encoding LTL formulas in a manner that compiles more quickly but requires the same or more time to model check than the equivalent SPOT-encoding.

6.5   Experimental Results

Our experiments demonstrate that the new Promela `never` claims we have introduced significantly improve the translation of LTL safety formulas into explicit automata, as measured by model checking time. While it is sometimes the case than many, or all, of our encodings incur less model checking time than SPOT's encoding, we found that one of our encodings is always best: `front_det_switch_min_abr_ea_state_fin`. We can consistently improve on the model checking time required for SPOT encodings by using `front_det_switch_min_abr_ea_state_fin` encoded `never` claims instead. Therefore, we recommend an optimal encoding approach of encoding LTL formulas known to be safety properties with our `front_det_switch_min_abr_ea_state_fin` encoding and all others with SPOT.

It is interesting to note that we found certain encoding aspects to be always better. This helps explain why the `front_det_switch_min_abr_ea_state_fin` encoding is always the fastest: it is the encoding that combines all of the fastest `never` claim components. In particular, we found the following trends to hold for encodings where all other encoding components were the same.

- Deterministic (`det`) `never` claims are faster than determinized-on-the-fly (`nondet`)

`never` claims.

- Finite acceptance (`fin`) is faster than infinite acceptance (`inf`).

- State labels (`state`) are faster than state numbers (`number`).

- Edge abbreviation (`ea`) always equates to better performance.

- There seems to be a positive correlation between the code size of a given `never` claim and the required model checking time: smaller code runs faster. For example, state labels are faster than state numbers and they generally require less code. Encodings with Boolean-formula-labeled transitions are generally faster than those requiring calculation of the (`system_state_index`). The memory table encodings, which include large table declarations in addition to the computation of the `system_state_index` to perform the table look-up are the least scalable; Spin won't even compile them for reasonably-sized formulas. Code size is important.

6.5.1 Sometimes Deterministic Automata Are Much Better Than Nondeterministic Automata

For some benchmarks, we found that all of our encodings, whether they determinized $\mathcal{A}^d$ up front or on the fly, required less model checking time than the equivalent nondeterministic SPOT `never` claims.[5] For example, for the iprot and sliding window benchmarks, pictured in Figures 6.2(a) and 6.2(b), all of our new encodings performed better than SPOT, though our `front_det_switch_min_abr_ea_state_fin` encoding was best.

---

[5]Note that not all SPOT `never` claims are nondeterministic; for other benchmarks SPOT produced deterministic `never` claims.
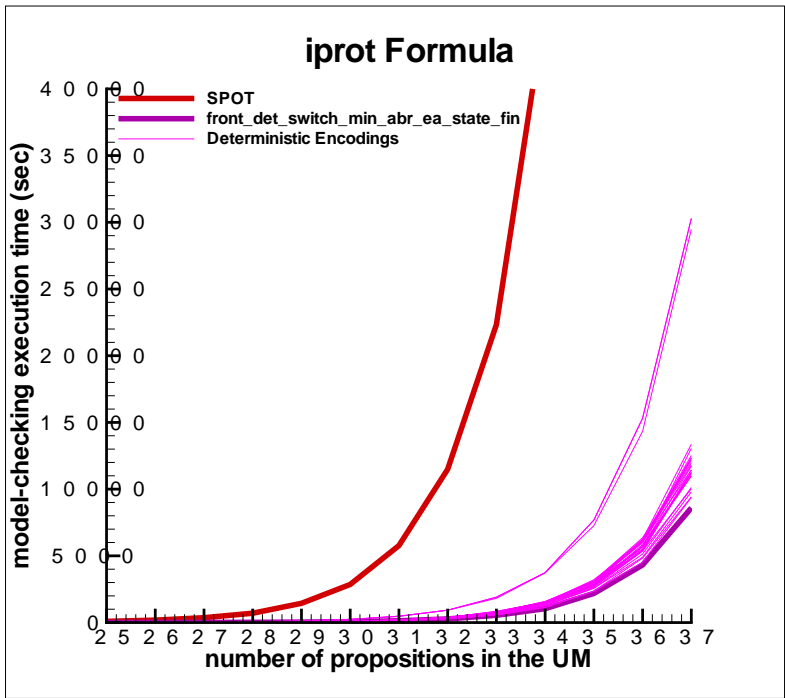
(a) Benchmarks for the iprot specification.
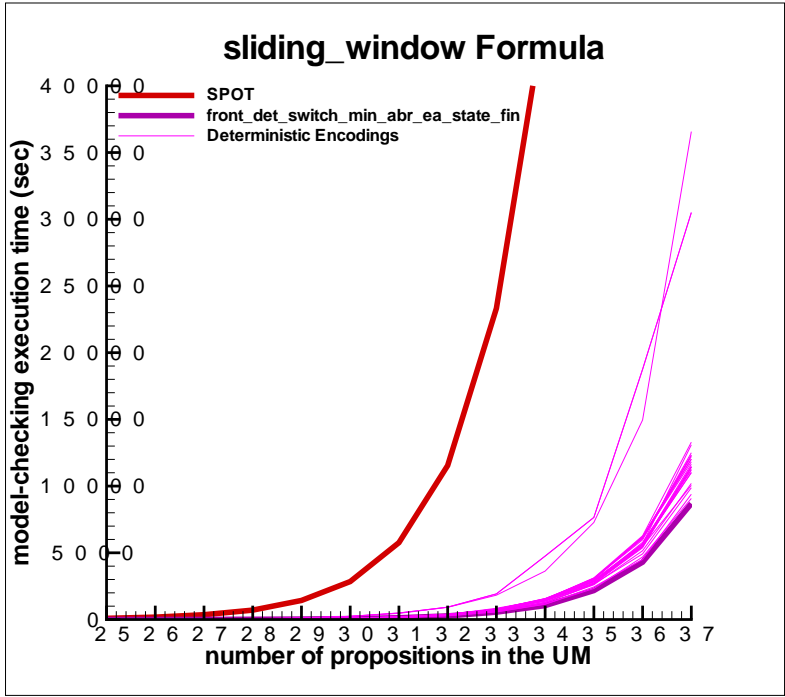
(b) Benchmarks for the sliding window specification.

Figure 6.2 : Model scaling benchmarks, showing the model-checking execution times based on the number of propositions in the UM.

Deterministic encodings can result in significant improvements in model checking performance by reducing calls to the internal nested depth-first search algorithm in the model checker; see Chapter 2.8. Take for example the AccidentallySafe benchmark, which encodes the formula $\Box(q \vee X\Box p) \wedge \Box(r \vee X\Box\neg p)$. The SPOT encoding for the corresponding never claim appears in Listing 6.11. As we increase the size of the universal model, the time required to model check this never claim increases exponentially.

```
1  never { /* F((!p1 & XF!p0) | (!p2 & XFp0)) */
2  T0_init:
3     if
4     :: (!(p2)) -> goto accept_S2
5     :: ((1)) -> goto T0_S3
6     :: (!(p1)) -> goto accept_S4
7     fi;
8  accept_S2:
9     if
10    :: ((p0)) -> goto accept_all
11    :: (!(p0)) -> goto T0_S6
12    fi;
13 T0_S3:
14    if
15    :: (!(p2)) -> goto accept_S2
16    :: ((1)) -> goto T0_S3
17    :: (!(p1)) -> goto accept_S4
18    fi;
19 accept_S4:
20    if
21    :: (!(p0)) -> goto accept_all
22    :: ((p0)) -> goto T0_S7
23    fi;
24 T0_S6:
25    if
26    :: ((p0)) -> goto accept_all
27    :: (!(p0)) -> goto T0_S6
28    fi;
29 T0_S7:
30    if
31    :: (!(p0)) -> goto accept_all
32    :: ((p0)) -> goto T0_S7
33    fi;
34 accept_all:
```

```
35    skip
36  }
```

Listing 6.11: Illustrating the SPOT `never` claim for the AccidentallySafe formula.

However, if we encode this same `never` claim deterministically, the time required to model check this `never` claim remains near zero as we increase the size of the universal model. For comparison, the `front_det_switch_min_abr_ea_state_fin` encoding of the AccidentallySafe benchmark appears in Listing 6.12. Why does this encoding take so little time to model check? We can answer this question by examining the initial state, `init_S1`, in Listing 6.12. In this case, Spin initially explores the valuation where the variables $p0$, $p1$, and $p2$ are false, in which case this `never` claim transitions directly to `done`, causing Spin to skip the NDFS in the emptiness check. It is the NDFS that causes the SPOT `never` claim to require exponentially increasing time to model check: note that the initial state in Listing 6.11 has no equivalent deterministic path to termination.

```
1  /*LTL formula: (!([](p1 | (X [] p0)) & [](p2 | (X ([] ! p0)))))*/
2  never {
3  init_S1:
4    atomic {
5      if
6        :: (p2 && !p1 )
7          -> goto   S2;
8        :: (p1 && p2 )
9          -> goto   init_S1;
10       :: (p1 && !p2 )
11         -> goto   S0;
12       :: else -> goto done;
13     fi;
14   }
15 S0:
16   atomic {
17     if
18       :: (p1 && !p0 )
19         -> goto   S0;
20       :: else -> goto done;
21     fi;
22   }
```

```
23  S2:
24    atomic {
25      if
26        :: (p0 && p2 )
27          -> goto   S2;
28        :: else -> goto done;
29      fi;
30    }
31  done: /*signal property violation by landing here*/
32    skip;
33  }
```

Listing 6.12: Illustrating the front_det_switch_min_abr_ea_state_fin never claim for the AccidentallySafe formula.

6.5.2 We are consistently faster than SPOT.

Figures 6.3(a) and 6.3(b) show the median model checking times of randomly-generated safety formulas, both completely randomly generated, as in Figure 6.3(a) and syntactically safe randomly generated formulas, as in Figure 6.3(b). As these figures show, our median model checking times over all non-trivial randomly generated formulas were consistantly lower than for SPOT encodings.

Since we call SPOT as a step in our encoding, our median automaton generation times were always higher than SPOT but, despite our inefficient implementation, were consistently dwarfed by model checking times. For example, for the set of 5-variable, length 20 random formulas included in Figure 6.3(a), our median LTL-to-never claim time was 0.12 seconds, median Promela-to-C time was 0.0 seconds, median C-to-binary time was 0.23 seconds, and median model-checking time was 51.08 seconds. For SPOT encodings, the median LTL-to-never claim time was 0.01 seconds, median Promela-to-C time was 0.0 seconds, median C-to-binary time was 0.25 seconds, and median model-checking time was 1582.385 seconds. So, our CHIMP-Spin tool required in the median case about 12 times as much time to create a never claim as SPOT, though it is notable that this time
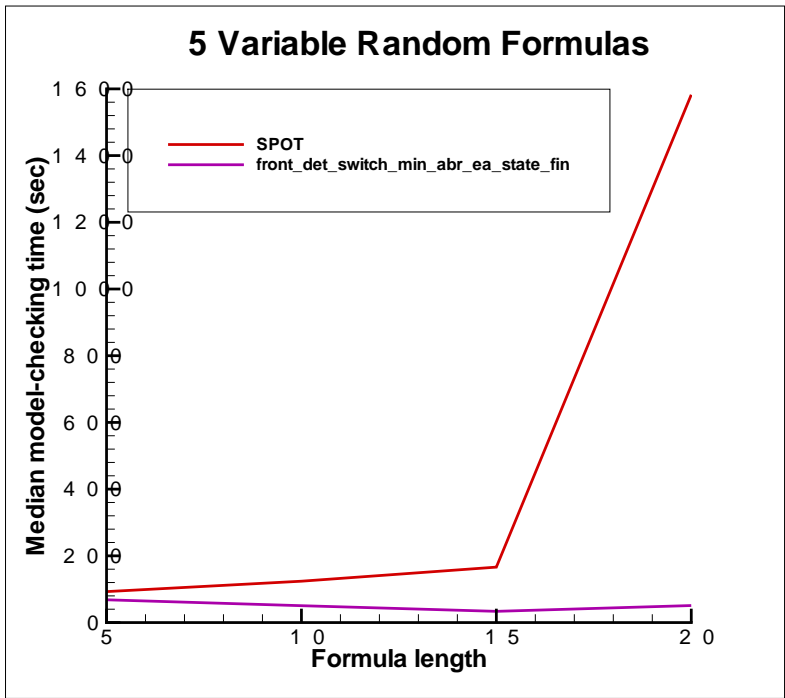
(a) Median model-checking times for 5 variable random formulas.
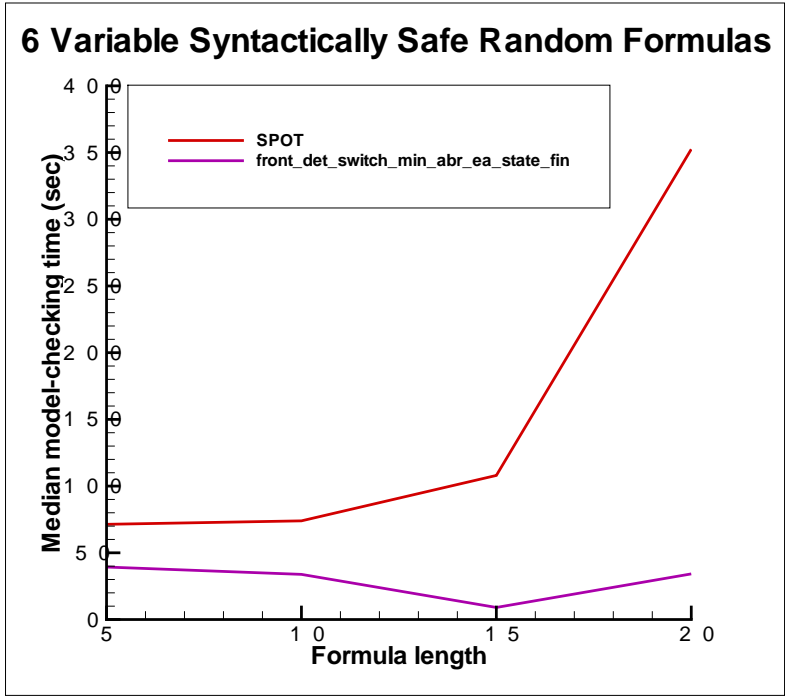
(b) Median model-checking times for 6 variable syntactically safe formulas.

Figure 6.3 : Graphs of median model-checking times for both categories of randomly-generated formulas, showing that our median model checking times were consistently lower than SPOT.
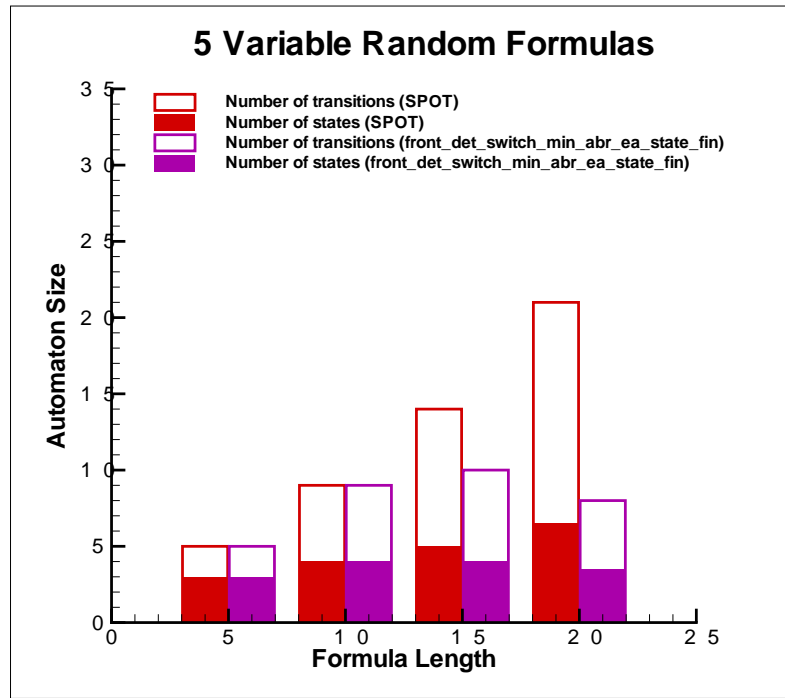
(a) Automaton size for the 5 variable random formulas.
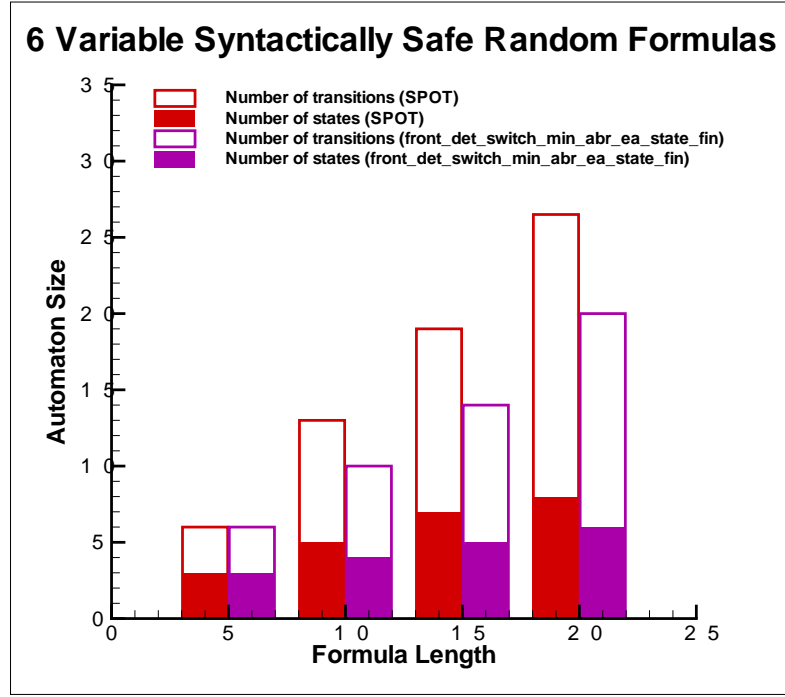
(b) Automaton size for the 6 variable syntactically safe formulas.

Figure 6.4 : Graphs of automaton size for both categories of randomly-generated formulas, showing that our automata sizes compared to SPOT.

was still a fraction of a second. This trend holds for syntactically safe random formulas as well. For example, for the set of 6-variable, length 20 random formulas included in Figure 6.3(b), our median LTL-to-`never` claim time was 0.13 seconds, median Promela-to-C time was 0.01 seconds, median C-to-binary time was 0.26 seconds, and median model-checking time was 34.195 seconds. For SPOT encodings, the median LTL-to-`never` claim time was 0.01 seconds, median Promela-to-C time was 0.0 seconds, median C-to-binary time was 0.26 seconds, and median model-checking time was 352.475 seconds.

Note that BRICS experienced some errors when encoding some of our randomly generated formulas. These cases were rare enough as to not affect our median results, i.e. for the set of 500 5-variable, 15-length random formulas graphed in Figure 6.3(a), BRICS experienced nine errors. However, the occurrence of these errors during our LTL-to-`never` claim phase combined with the consistent time difference between CHIMP-Spin and SPOT for this phase mean that the *total time* required by CHIMP-Spin for LTL compilation plus model checking was not always less than SPOT for randomly-generated formulas with small model-checking times.

It is interesting to note that the difference in model checking time is not directly correlated with other statistics we measured, such as the length of counterexamples returned for formula violations. Across all of the randomly-generated formula benchmarks, we found that the number of states and the lengths of counterexamples associated with our `front_det_switch_min_abr_ea _state_fin` never claims and with SPOT's were usually very close, within a few states of each other. However, in general, the number of transitions had a higher variance between these two encodings; in the median cases, we ended up with less than or equal to the number of transitions in the equivalent SPOT `never` claim, as shown in Figures 6.4(a) and 6.4(b). These two figures show the median number of states and transitions for the two benchmarks in Figures 6.3(a) and 6.3(b), respectively.

While we found our winning encoding could always reduce the time required for model
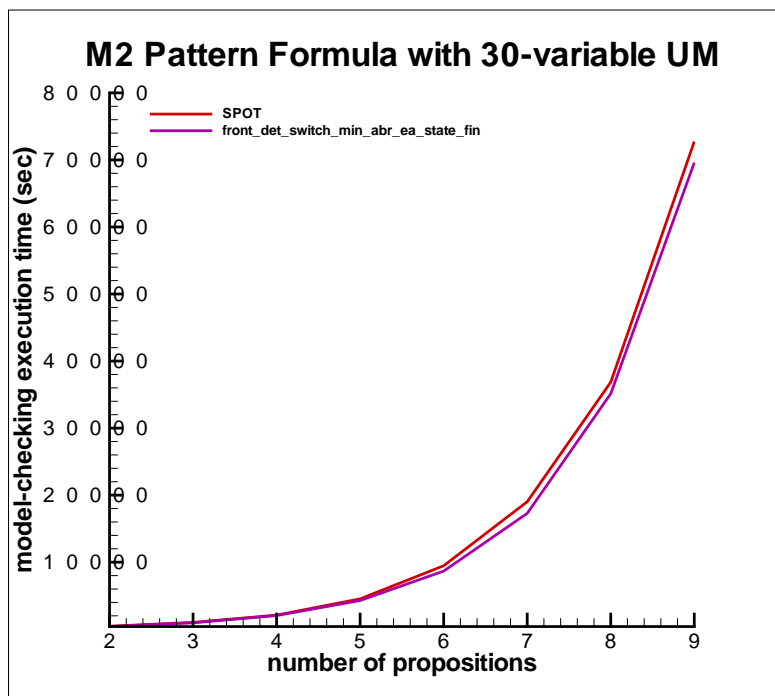
(a) Model-checking time for the M_2 pattern scaled formula benchmark: From the $M_2(n)$ pattern equation in Chapter 3.6.2, it is easy to see that formula length is $2.5n(n-1)$.
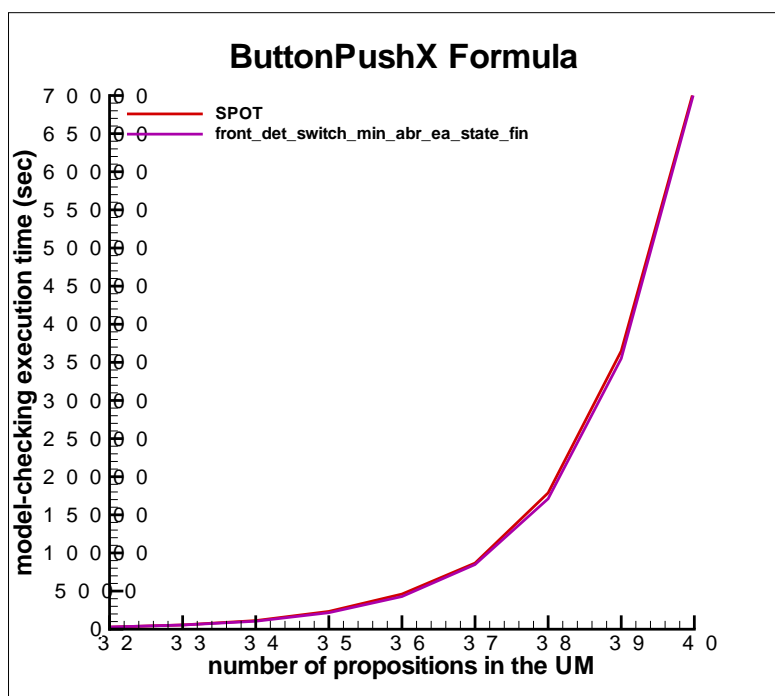
(b) Model-checking time for the ButtonPushX scaled model benchmark.

Figure 6.5 : While our best encoding always incurred the lowest model checking times, for some instances of both formula scaling and model scaling benchmarks, our improvement over SPOT was small.

checking over that of the SPOT encoding for the same formula, we did find some benchmarks where the difference was small. For example, for the M2 benchmark displayed in Figure 6.5(a), the model checking time for never claims encoded by our `front_det_switch_min_abr_ea_state_fin` encoding is consistently slightly better than SPOT.

Out of all of our benchmarks, the ButtonPushX benchmark displayed the smallest difference between our encoding and SPOT, as shown in Figure 6.5(b). For example, for the 36-variable universal model, the SPOT `never` claim took 4606.94 seconds, or roughly 77 minutes whereas our `never` claim took 4281.22 seconds, or roughly 71 minutes, only 6 minutes less. Still, our `front_det_switch_min_abr_ea_state_fin` encoding enabled Spin to scale to model check a 40-variable model whereas model checking the SPOT `never` claim timed out at 39 variables.

Note that for model-scaling benchmarks, the difference in compile time between SPOT and CHIMP-Spin was negligible. For Figures 6.2(a), 6.2(b), and 6.5(b) SPOT takes approximately 0.01 seconds to construct a `never` claim from an LTL formula while our inefficient CHIMP-Spin code takes about 0.1 seconds; for all data points shown in all three figures, the Promela-to-C compilation time was too small to measure and the C-to-binary compilation time was approximately 0.2 seconds. For formula-scaling benchmarks total compile time scaled but was still dwarfed by model-checking time; for example, CHIMP-Spin spent about 0.2 seconds creating a `never` claim for the 9-variable benchmark shown in Figure 6.5(a).

6.6 Discussion

This chapter brought attention to the benefit of deterministic encoding of finite automata for safety properties. We defined novel encodings of safety LTL properties as `never` claims and showed that one encoding consistently leads to faster model checking times than the

state-of-the-art SPOT encoding or any of our other new encodings. Therefore, we recommend a two-encoding optimized front-end to the Spin model checker: use SPOT for the optimal encodings of non-safety properties and use our new `front_det_switch_min_abr_ea_state_fin` encoding for known safety properties.

The size of the product automaton is not an accurate measure of performance when there is a property violation since the decision of which transitions to explore first plays a more crucial role. When the `never` claim is nondeterministic, the order of searching transitions can make a big difference even when it does not change the semantics. Since Spin search transitions in a certain order, shuffling the lines in a nondeterministic `never` claim can have a dramatic impact on model-checking time. We demonstrated this problem can be avoided in the case of safety properties via our winning deterministic encoding with finite acceptance conditions. Our winning encoding is the combination of all of the best-performing encoding components. In this way, our approach is *extensible*: others may find other encoding components that are compatible with our winning components to improve the model checking performance of `never` claims in the future.

Our results concur with the findings in Chapter 4.6 and [228] that automaton size, in terms of number of automaton states, number of transitions, or both is not a reliable indicator of the time required by the Spin model checker to check the specification automaton. We found that sets of `never` claims representing different encodings of the same LTL formula, with the same number of states and transitions, performed vastly differently from each other in terms of model checking time. Interestingly, we did note a positive correlation between the size and simplicity of the code comprising the `never` claim and the model checking time. It seems to be the case that smaller, simpler `never` claims for the same deterministic automaton tend to be faster to model check, whether they are smaller because the code is more efficient or because they have fewer states or transitions. The question of whether these results extend to other domains (i.e. symbolic model checking)

is an interesting direction for a future investigation.

Chapter 7

Conclusion

As safety-critical systems continue to grow in terms of both their number and their size, so too grows the need to provide tools for rigorous verification. In response to the scale and complexity of these systems and their specifications, it it highly likely that new techniques for efficiently reasoning about them will continue to be developed. Future tools for LTL-to-automata translation, whether they create explicit or symbolic automata, will need to be both correct and scalable. We have provided a solid basis on which to build future tools in terms of both addressing their performance, and creating better, more scalable translations.

In this chapter, we recap the work we have presented in this dissertation, briefly examine its impact, and the discuss potential avenues for future research. In Section 7.1 we review our major contributions and their impact to date. We address the future for each of the areas of research we have contributed to in Section 7.2 before concluding with some final remarks in Section 7.3.

7.1   Contribution Review

In this thesis, we address formal temporal specifications expressible in the popular logic LTL. In accordance with the modern paradigm of property-based design that has been adopted in response to the need for design-phase checks on today's dependable systems, we advocated for the adaptation of a new sanity check. We argued that each specification, its negation, and the conjunction of all specifications should be checked for satisfiability as an essential first step of the verification process. We demonstrated utilizing LTL model

checkers to perform this test to help ensure we have specified the behaviors we intended, before the system described is built to these requirements.

We established a set of rigorous benchmarks to enable more accurate assessment of LTL-to-automaton translation performance in new ways, evaluating efficiency, scalability, and correctness. Our benchmarks are valuable both in the context of LTL satisfiability checking and LTL model checking. We released these benchmarks, along with benchmark-generating code that produces our scaled benchmarks, on the web for widespread use in evaluating new LTL-to-automaton translation algorithms.[1]

We presented a thorough examination of the state of the art in LTL-to-automaton translation, both for explicit and symbolic automata. We accomplished this by analyzing, in depth, essentially all tools for LTL-to-automaton translation that were available at the time of our study, measuring a wide range of statistics on each one. Where previous studies used automaton size as a proxy for measuring performance, we demonstrated that these two statistics were not correlated as closely as previously thought.

We created 29 novel encodings for LTL formulas as symbolic automata, where there had only ever been one before. We introduced a new multi-encoding approach, running 23 of these encodings in parallel, and showed that this approach consistently bests the current state of the art, performing up to exponentially better at LTL satisfiability checking. Our LTL-to-symbolic automata translation tool, PANDA, has been released so that others may add to this extensible framework.[2]

For the most common type of specifications, safety properties, we introduced 26 novel encodings of LTL formulas as explicit automata, exploiting the inherent determinism in these properties in order to improve the performance of the model checking step. Indeed,

---

[1]http://ti.arc.nasa.gov/m/profile/kyrozier/benchmarking_scripts/
benchmarking_scripts.html

[2]http://ti.arc.nasa.gov/m/profile/kyrozier/PANDA/PANDA.html

we have shown we can consistently reduce the time required to perform model checking using safety specifications by replacing the current state of the art encoding by our one best encoding, which we call `front_det_switch_min_abr_ea_state_fin`.

7.1.1   Impact

The world of Linear Temporal Logicians was clearly hungry for a set of quality benchmarks for fairly evaluating algorithms that reason about LTL formulas. Our benchmarks have been downloaded and used to evaluate all relevant algorithms that we know of since their initial publication; see for example [57, 232, 58, 233, 234].

Our evaluation of LTL-to-automaton algorithms has inspired others to more carefully choose the algorithms and tools they use for this critical step in the verification process. The author of SPOT utilized our benchmarks in fixing the rare bug we uncovered in his tool; our benchmarks continue to be used to demonstrate the quality of successive versions of SPOT. Since our original study, others have cited our work as an influence in their choice of the two best explicit translators, SPOT and LTL2BA, for LTL-to-automata [235, 236, 237, 1, 238, 239, 240]. Similarly, this work influenced the choice of SMV and symbolic LTL-to-automata for LTL satisfiability checking [241, 242]. It has also influenced others to adopt satisfiability checking as a standard sanity check [50, 243, 244].

Only months after our study was published, our work on a multi-encoding approach for symbolic LTL satisfiability checking has already been built upon. Others advocated for a related multi-encoding approach for LTL satisfiability [234]. PANDA has also already been used in a full-scale real-world verification project to enable better scalability for specification debugging for an automated air traffic control system at NASA [245].

7.2 Future Work

Perhaps the most obvious direction for future work is devising new explicit or symbolic encodings of LTL formulas, evaluating them utilizing our standardized benchmarks, and adding the best-performing ones to our current tools to further improve the empirical performance of LTL satisfiability and model checking. While we certainly expect to see work in this direction in the near future (and indeed others have already started building upon our work to create additional encodings), there are also many other future directions worth pursuing.

7.2.1 Future Work on LTL-to-Symbolic Automata

LTL symbolic model checking is ideal for the verification of *reactive systems*, or those systems that operate within a dynamic environment such as concurrent programs, embedded and process control programs, and operating systems. LTL allows us to naturally and organically specify reactive systems in terms of their ongoing behavior. By shifting the focus of the state explosion problem from the size of the state space (as in explicit-state model checking) to the size of the BDD representation, symbolic model checking can be used to verify much larger systems. Though the time complexity bottleneck of the LTL symbolic model-checking algorithm is arguably the translation step from φ to $A_{\neg\varphi}$, there are numerous facets of the algorithm worth optimizing, not all of which were mentioned above. Given the promise of this method of formal verification and the real-world verification successes so far, optimization of virtually every segment of this algorithm is a possible subject for future research.

While our idea of exploiting the inherent determinism of safety properties to create new encodings to speed up model checking performance was very successful in the explicit model checking domain, it remains an open question whether a similar feat might

be possible in the symbolic model checking domain. For example, it is not clear whether there are equivalent symbolic encodings of LTL safety properties to the explicit encodings we presented in Chapter 6. If we can devise such encodings, would they provide a similar performance boost in the domain of symbolic model checking? And if so, would it be worthwhile to expand our PANDA tool to add these new encodings? These are questions we plan to pursue in the future.

We plan to utilize and build upon our tool, PANDA, in additional ways. Since our new symbolic encodings triggered such a dramatic, sometimes exponential, increase in performance for symbolic LTL satisfiability checking, it is worthwhile to conduct a rigorous study to answer the question of whether we can similarly boost LTL symbolic model checking performance. We may also be able to expand PANDA to produce symbolic automata in Promela as well as the current output in SMV language. This could potentially lead to speed-ups for explicit LTL satisfiability checking and explicit model checking with Spin.

Perhaps the lessons learned from our study and creation of PANDA may spur encoding improvements for other related logics as well. For example, we plan to adapt PANDA to efficiently encode fragments of the new logic CSSL, a logic for specifying conditional scenarios [246]. CSSL formulas are comprised of pairs of LTL formulas connected by CSSL operators at the top level. It may be the case that we can reduce the time required to model check CSSL formulas in practice by utilizing PANDA's arsenal of symbolic automata encodings.

7.2.2   Future Work on LTL Model Checking of Safety Properties

In practice, a significant fraction of the complexity of the best LTL-to-explicit automaton translation presented in Chapter 6 is attributed to accommodating artifacts of the BRICS automaton tool. We translate into and out of formats readable by BRICS, modify the transition labels created by BRICS, prune trap states from the BRICS automaton, and catch cases

where BRICS scalability limitations prevent us from completing the LTL-to-automaton translation. Now that we know the resulting `never` claim is advantageous, in the future we could produce it more efficiently by implementing a homegrown version of BRICS to better serve our needs.

Other future directions for this work include lobbying the creators of the model checker Spin to enable Promela specifications to be compiled once and used for model checking many times. This, combined with our more efficient `never` claims could significantly reduce the time engineers spend using Spin for system verification. We also envision adding an automated dual-encoding front-end to Spin that could automatically check if the input LTL formula is a safety formula (i.e. checking for syntactic safety can be done very efficiently) and then create the `never` claim using our `front_det_switch_min_abr_ea_state_fin` encoding if this is the case and using SPOT if it is not.

7.3   Concluding Remarks

As more and more functions in our everyday lives become automated, the need for formal verification rises as well; we entrust our safety to these systems. It becomes ever more important that the methods we use for formal verification are both *correct* and *scalable*. However, previous LTL-to-automaton translation algorithms did not achieve these goals, as our extensive survey of LTL-to-automaton translation tools showed. We have changed the way LTL-to-automaton translation algorithms are evaluated and provided more scalable translations in both the explicit and the symbolic domains. We have pushed the state of the art and provided a solid foundation on which we and others can build (indeed on which many have already begun building).

We have brought attention to promising but previously neglected areas of LTL-to-automaton translation for specification debugging via satisfiability checking and for model

checking. We have established a de facto standard set of benchmarks that aid in the development of new, correct, LTL-to-automaton translations and used these benchmarks in the development of our own translation algorithms. We have demonstrated a significant improvement in scalability over the previous state of the art. Our approaches to both explicit and symbolic LTL-to-automaton translation are *extensible*. We expect others to build upon them in the future and improve upon our empirical results as we have done for those that came before us. We believe our approaches to explicit and symbolic LTL-to-automaton translation provide practical methods to tackle many current verification problems, and provide a foundation upon which to develop new research that has the potential to provide understanding of even more complex systems, allowing system architects to efficiently perform specification debugging and model checking on future safety-critical systems.

Appendix A

Benchmarking Scripts

This appendix contains further details of the code used to generate all of the benchmarks described in Chapter 3. All of the software described here is available for download under a NASA Software Agreement[1] from `http://ti.arc.nasa.gov/m/profile/` `kyrozier/benchmarking_scripts/benchmarking_scripts.html`.

A.1 Explicit Tool Models

To benchmark LTL formula translations, we use for the SPIN model a *universal* Promela model, enumerating all possible traces over *Prop*. For example, when $Prop = \{a, b\}$, (and therefore the number n of variables is 2), the Promela model is:

```
/*File: 2vars.pml*/

bool A,B;

/* define an active procedure to generate values for A and B */

active proctype generateValues()
{ do
   :: atomic{ A = 0; B = 0; }
   :: atomic{ A = 0; B = 1; }
   :: atomic{ A = 1; B = 0; }
   :: atomic{ A = 1; B = 1; }
  od }
```

---

[1] The full text of the NASA Software Agreement can be found here: `http://opensource.arc.` `nasa.gov/license.jsp`.

We use the `atomic{}` construct to ensure that the Boolean variables change value in one unbreakable step. Note that the size of this model is exponential in the number of atomic propositions. It is also possible to construct a model that is linear in the number of variables like this:

```
/*File: 2vars.pml*/

bool A,B;
active proctype generateValues()
{
  do
   :: atomic{
      if
         :: true -> A = 0;
         :: true -> A = 1;
      fi;
      if
         :: true -> B = 0;
         :: true -> B = 1;
      fi;
   }
  od
}
```

The variables in *Prop* are defined in the accompanying file `2vars.define`, which is prepended to the never-claim we are testing.

These two files, *n*vars.pml and *n*vars.define, are generated automatically for any given value of *n* by the following scripts:

Script Name	Variable Set	Recommended Input Formulas
generateValues.pl	Prop = a, b, c, ..., aa, bb, cc, ...	Use with random and counter formulas. (Flags designate exponential vs linear construction.)
generatePvalues.pl	Prop = p1, p2, p3, ...	Use with pattern formulas.
generateAlphaPvalues.pl	Prop = po, pt, ph, ...	Use with pattern formulas, for tools that don't allow numbers in variable names

Here is an example of how to use these scripts to run SPIN on a generated Promela never-claim with a universal model:

```
#Precede the formula with a 'next time' operator to skip
#  SPIN's nondeterministic assignment upon declaration
$LTL_to_automata_tool_binary ``(X($input_formula))" >& $temp
generateValues.pl $n
cp -f $nvars.define pan.ltl
cat $temp >> pan.ltl
rm -f $temp
cp -f $nvars.pml pan_in
spin -a -X -N pan.ltl pan_in
gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=128 -DXUSAFE -DNOFAIR  pan.c
./pan -v -X -m10000 -w19  -a -c1
```

A.2 Symbolic Tool Models

We can also benchmark LTL formula modeling with CadenceSMV and NuSMV. To check whether a LTL formula φ is satisfiable, we model check $\neg\varphi$ against a universal SMV model. For example, if $\varphi = (X(a))$, we provide the following input to NuSMV:

```
MODULE main
```

```
VAR
  a : boolean;
  b : boolean;
  c : boolean;
LTLSPEC !(X(a=1 ))
FAIRNESS
  1
```

(The "FAIRNESS" line varies with versions of NuSMV. The CadenceSMV model is very similar.) SMV negates the specification, $\neg\varphi$, symbolically compiles φ into A_φ, and conjoins A_φ with the universal model. If the automaton is not empty, then SMV finds a fair path, which satisfies the formula φ. In this way, SMV acts as both a symbolic compiler and a search engine.

We generate symbolic tool universal models on the fly since the code is very simple. For example, here is the perl code to generate the NuSMV model above:

```perl
#create a file to input the LTL model:
@N = qw(a b c); #array of variable names
$tempModel = "temp.smv";
open(MFILE, ">$tempModel") or die "Could not open $tempModel: $!";
print MFILE "MODULE main\n";
print MFILE "  VAR\n";
for ($i = 0; $i < $n; $i++) {
   if ($scaleable_formula == 0) { #random and counter formulas
      print MFILE "    $N[$i] : boolean;\n";
   } #end if
   else { #pattern formulas
      $iplus1 = $i + 1;
      print MFILE "    p${iplus1} : boolean;\n";
   } #end else
} #end for
print MFILE "  LTLSPEC !($formula)\n";
print MFILE "  FAIRNESS\n";
print MFILE "    1\n";
close(MFILE) or die "Could not close $tempModel: $!";
```

A.3 Input Formulas

In [56], we benchmarked the tools against three types of scalable formulas: random formulas, counter formulas, and pattern formulas. Scalability played an important role in our experiment, since the goal was to challenge the tools with large formulas and state spaces.

A.3.1 Counter Formulas

Pre-translation rewriting is highly effective for random formulas, but ineffective for structured formulas [186, 136]. To measure performance on scalable, non-random formulas we tested the tools on formulas that describe n-bit binary counters with increasing values of n. These formulas are irreducible by pre-translation rewriting, uniquely satisfiable, and represent a predictably-sized state space. Correctness is easily determined by checking for precisely the unique counterexample for each counter formula. We tested four constructions of binary counter formulas, varying two factors: number of variables and nesting of $\mathcal{X}$'s.

We can represent a binary counter using two variables: a counter variable and a marker variable to designate the beginning of each new counter value. Alternatively, we can use 3 variables, adding a variable to encode carry bits, which eliminates the need for $\mathcal{U}$-connectives in the formula. We can nest $\mathcal{X}$'s to provide more succinct formulas or express the formulas using a conjunction of unnested $\mathcal{X}$-sub-formulas.

Let b be an atomic proposition. Then a computation π over b is a word in $(2^{\{0,1\}})^\omega$. By dividing π into blocks of length n, we can view π as a sequence of n-bit values, denoting the sequence of values assumed by an n-bit counter starting at 0, and incrementing successively by 1. To simplify the formulas, we represent each block $b_0, b_1, \ldots, b_{n-1}$ as having the most significant bit on the right and the least significant bit on the left. For example, for $n = 2$ the b blocks cycle through the values 00, 10, 01, and 11. For technical convenience, we

use an atomic proposition *m* to mark the blocks. That is, we intend *m* to hold at point *i* precisely when $i = 0 \mod n$.

For π to represent an *n*-bit counter, the following properties need to hold:

1) The marker consists of a repeated pattern of a 1
   followed by n-1 0's.

2) The first n bits are 0's.

3) If the least significant bit is 0, then it is 1 n steps later
   and the other bits do not change.

4) All of the bits before and including the first 0
   in an n-bit block flip their values in the next block;
   the other bits do not change.

For $n = 4$, these properties are captured by the conjunction of the following formulas:

```
1.  (m) && ( [](m -> ((X(!m)) && (X(X(!m)))

                              && (X(X(X(!m))))

                              && (X(X(X(X(m))))))))))

2.  (!b) && (X(!b)) && (X(X(!b))) && (X(X(X(!b))))

3.  []( (m && !b) ->

        ( X(X(X(X(b)))) &&

          X ( ( (!m) &&

             (b -> X(X(X(X(b))))) &&

             (!b -> X(X(X(X(!b)))))  ) U m ) ) )

4.  [] ( (m && b) ->

        ( X(X(X(X(!b)))) &&

          (X ( (b && !m && X(X(X(X(!b))))) U

                (m ||

                 (!m && !b && X(X(X(X(b)))) &&
```

```
X( ( !m && (b -> X(X(X(X(b)))))  &&

        (!b -> X(X(X(X(!b)))))  )  U

     m ) ) ) ) ) ) )
```

Scalable counter formulas of this form are automatically generated by LTLcounter.pl.[2]

```
Usage: LTLcounter.pl number_of_bits
```

Note that this encoding creates formulas of length $O(n^2)$. A more compact encoding results in formulas of length $O(n)$. For example, we can replace formula (2) above with:

```
2. ((!b) && X((!b) && X((!b) && X(!b))))
```

Scalable counter formulas of this form, with nested X-operators, are automatically generated by LTLcounterLinear.pl.[3]

```
Usage: LTLcounterLinear.pl number_of_bits
```

We can eliminate the use of $\mathcal{U}$-connectives in the formula by adding an atomic proposition c representing the carry bit. The required properties of an n-bit counter with carry are as follows:

A 4-bit Binary Counter

m	0001	0001	0001	0001	0001	0001	0001	0001	0001	0001
b	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
c	0000	0001	0000	0011	0000	0001	0000	0111	0000	0001

m	0001	0001	0001	0001	0001	0001	0001	...
b	1010	1011	1100	1101	1110	1111	0000	...
c	0000	0011	0000	0001	0000	1111	0000	...

[2]http://ti.arc.nasa.gov/m/profile/kyrozier/benchmarking_scripts/
LTLcounter.pl.zip

[3]http://ti.arc.nasa.gov/m/profile/kyrozier/benchmarking_scripts/
LTLcounterLinear.pl.zip

To describe the behavior of the counter, we describe six properties and AND them together.
For example, here is the program output for a 4-bit counter:

```
1) The marker consists of a repeated pattern of a 1
   followed by n-1 0's.
2) The first n bits are 0's.
3) If m is 1 and b is 0 then c is 0 and n steps later b is 1.
4) If m is 1 and b is 1 then c is 1 and n steps later b is 0.
5) If there is no carry,
   then the next bit stays the same n steps later.
6) If there is a carry,
   flip the next bit n steps later and adjust the carry.
```

For $n = 4$, these properties are captured by the conjunction of the following formulas.

```
1. (m) && ( [](m -> ((X(!m)) && (X(X(!m)))

                             && (X(X(X(!m))))

                             && (X(X(X(X(m)))))))))
2. (!b) && (X(!b)) && (X(X(!b))) && (X(X(X(!b))))
3. [] ( (m && !b) -> (!c && X(X(X(X(b)))))  )
4. [] ( (m && b)  -> (c && X(X(X(X(!b)))))  )
5. [] (!c && X(!m)) ->
       ( X(!c) && (X(b) -> X(X(X(X(X(b)))))) &&
                  (X(!b) -> X(X(X(X(X(!b))))))  )
6. [] (c -> ( ( X(!b) ->
                 ( X(!c) && X(X(X(X(X(!b)))))  ) ) &&
               ( X(c)  && X(X(X(X(X(b)))))  ) ))
```

Scalable counter formulas of this form, using a carry bit and no $\mathcal{U}$-operators, are auto-

matically generated by LTLcounterCarry.pl.[4]

```
Usage: LTLcounterCarry.pl number_of_bits
```

Scalable counter formulas using a carry bit and nested X-operators are automatically generated by LTLcounterCarryLinear.pl.[5]

```
Usage: LTLcounterCarryLinear.pl number_of_bits
```

A.3.2 Pattern Formulas

These scripts generate formulas from the eight classes of scalable formulas defined by [64] to evaluate the performance of model checking algorithms on temporally-complex formulas.

Run them like this:

```
formula.pl [-v] n
```

where n is the number of variables (i.e. the size) of the formula desired. The flag -v is for "verbose."

Patterns Used For LTL Satisfiability Checking

The following nine patterns are well-suited to benchmarking the performance of LTL-to-automaton translation algorithms.

$$E(n) = \bigwedge_{i=1}^{n} \Diamond p_i$$

$$U(n) = (\ldots (p_1 \, \mathcal{U} \, p_2) \, \mathcal{U} \, \ldots) \, \mathcal{U} \, p_n$$

---

[4] http://ti.arc.nasa.gov/m/profile/kyrozier/benchmarking_scripts/LTLcounterCarry.pl.zip

[5] http://ti.arc.nasa.gov/m/profile/kyrozier/benchmarking_scripts/LTLcounterCarryLinear.pl.zip

$$R(n) = \bigwedge_{i=1}^{n} (\square \Diamond p_i \lor \Diamond \square p_{i+1})$$

$$U_2(n) = p_1 \, \mathcal{U} \, (p_2 \, \mathcal{U} \, (\dots p_{n-1} \, \mathcal{U} \, p_n) \dots)$$

$$C_1(n) = \bigvee_{i=1}^{n} \square \Diamond p_i$$

$$C_2(n) = \bigwedge_{i=1}^{n} \square \Diamond p_i$$

$$Q(n) = \bigwedge (\Diamond p_i \lor \square p_{i+1})$$

$$S(n) = \bigwedge_{i=1}^{n} \square p_i$$

$$R_2(n) = (\dots (p_1 \, \mathcal{R} \, p_2) \, \mathcal{R} \, \dots) \, \mathcal{R} \, p_n$$

1. Eformula.pl

2. Uformula.pl

3. Rformula.pl

4. U2formula.pl

5. C1formula.pl

6. C2formula.pl

7. Qformula.pl

8. Sformula.pl

9. R2formula.pl

For model checking of safety properties

We evaluate LTL-to-automaton translation algorithms using three challenging scalable safety formula patterns. Note that $\overline{E}$ is the negation of the formula from [56, 64]. Recall from Chapter 3.2 that starred formulas are checked against universal models that set all variables to *true* first.

$$\overline{E}(n) = \neg\left(\bigwedge_{i=1}^{n} \mathcal{F}p_i\right),$$

$$X_1^*(n) = \bigvee_{i=1}^{n} (\Box Xp_i \vee X\Box p_{i+1}),$$

$$M_2(n) = \Box \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} (\neg(p_i \wedge p_j)).$$

1. Ebarformula.pl

2. X1formula.pl

3. M2formula.pl

A.3.3 Random Formulas

In order to cover as much of the problem space as possible, we use a script to generate sets of 500 randomly-generated formulas varying the formula length and number of variables as in [135]. It randomly generates sets of 500 formulas varying the number of variables, N, from 1 to 5, and the length of the formula, L, from 5 up to 200. It sets the probability of choosing a temporal operator $P = 0.5$ to create formulas with both a nontrivial temporal structure and a nontrivial Boolean structure. For temporal tests, formulas are also generated with $P = \frac{1}{3}$, $P = 0.7$, and $P = 0.95$. Other choices are decided uniformly. When generating a set of length L, every formula created is exactly of length L and not *up to L*, however,

these randomly-generated formulas are frequently reducible.

Here is the script used to generate random formulas in the style of [135]:

```
generateFormulas.pl
   With no arguments: generate a wide range of formula files
   With 3 arguments: generate files for specified L, N, P
      (order matters)
```

In our paper, [56], all of the random formulas we used were generated prior to testing, so each tool was run on the *same* formulas. To enable others to re-create our tests exactly, the set of formulas generated for our experiments are available online here: `http://ti.arc.nasa.gov/m/profile/kyrozier/benchmarking_scripts/formulas.zip`.

Random Safety Formulas

We generate random safety formulas similarly: we create sets of 500 m-length safety specifications over n atomic propositions, for m in $\{5, 10, 15, 20, 25\}$ and n in $\{2..6\}$. We set the probability of choosing a temporal operator $P = 0.5$.

   We have two ways of creating safety formulas randomly.

1. We generate syntactic safety formulas, allowing negation only directly before atomic propositions and limiting the temporal operators to $\{X, G, R\}$.

2. We generate each specification randomly over the full syntax of LTL. We then check if the generated specification represents a safety property using the SPOT command `ltl2tgba -O`, adding the specification to our test set if so and rejecting it if not. This method has the advantage that we are not restricted to syntactic safety formulas and the disadvantage that formula generation requires significant time since each formula must be checked for safety.

Here is the script used to generate random safety formulas:

```
Usage: generateSafetyFormulas.pl [-L#] [-P#] [-S]
                              proposition_list_file
        where
         -L#: indicates to create formulas of the given length
         -P#: gives the probability of choosing temporal operators
         -S: only produce syntactically safe formulas (i.e. NNF
             formulas using only X, G, R operators)
         proposition_list_file: (required) a file containing the
             atomic propositions to choose from, one per line
         (--help will produce a usage message)
         no inputs (default): just generate all lengths,
             combinations of operators and test for safety
             using SPOT


Outputs: a directory named (proposition_list_file)_safety_formulas,
            containing the desired files of safety formulas.
            All formulas generated are in NNF.
```

Appendix B

PANDA
(Portfolio Approach to Navigating the Design of Automata)

This appendix contains further details of the research code written for and used in the experiments described in Chapter 5. All of the software described here is available for download under a NASA Software Agreement from `http://ti.arc.nasa.gov/m/profile/kyrozier/PANDA`. The code described in this appendix is research prototype code that was run on the Shared University Grid at Rice (SUG@R), an Intel Xeon compute cluster.[1] This is not production-quality code; it may not run on other systems without modification.

B.1   PANDA C++ Code

At the lowest level, PANDA contains a module which creates one symbolic automaton encoding of one LTL formula, with the encoding and the output tool format dictated by command-line flags. This C++ code compiles into an executable with the following properties:

```
Input: an LTL formula

Grammar:
LTL_formula -> (formula)
formula ->
       -> formula & formula   //and
       -> formula | formula   //or
       -> ~ formula           //not
       -> formula -> formula  //implies
```

---

[1] `http://rcsg.rice.edu/sugar/`

```
          -> X formula            //next time
          -> formula R formula    //release
          -> G formula            //globally, always
          -> F formula            //in the future, eventually
          -> TRUE
          -> FALSE
          -> (formula)
          -> atomic_proposition


Output: a symbolic automaton model of the input formula (for NuSMV, Cadence SMV, or SAL)
              (see usage below)
          OR (with -pnnf) just print the negation normal form of the formula
          OR (with -pbnf) just print the boolean normal form of the formula
          OR (with -pnor) just print the formula with all R-operators replaced by ~(~aU~b)
              constructions

Dependencies:
   lex   //we used lex 2.5.35 distributed with RedHat
   yacc
   rgl2  //Graph triangulation implementation coded by the Kavraki Lab at Rice University.


Usage: PANDA ([(-c|-n|-s) (-nnf | -bnf) (-sloppy | -fussy) (-gba | -tgba)
               [(-mcs & (-max | -min | -random | -zero) | -lexm | -lexp
                 | -linear [-reverse]) -v]] | (-a | -pbnf | -pnnf | -pnor))
             LTL_formula_file
         where LTL_formula_file contains one line: a formula in the following syntax.
         Operators (in order of precedence):
         ~ (not), & (and), | (or), -> (implication)
         plus the following temoral operators:
         X p is true at time t if p is true at time t + 1.
         G p is true at time t if p is true at all times t' >= t.
         F p is true at time t if p is true at some time t' >= t.
         p U q is true at time t if q is true at some time t' >= t,
           and for all times < t' but >= t, p is true.
         p R q is true at time t if either q is true at time t and all times following t,
           or if p is true at some time t' >= t, and for all times between t and t',
           inclusive, q is true.
         Optional Flags:
           -pbnf to ONLY print formula in Boolean Normal Form and exit OR
           -pnnf to ONLY print formula in Negation Normal Form and exit OR
           -pnor to ONLY print formula without R-operators and exit OR
           -n for NuSMV-style comments in the symbolic automaton OR
           -c (default) for CadenceSMV-style comments in the symbolic automaton OR
           -s for SAL-style symbolic automaton AND
           -nnf to convert the formula to NNF in the parse tree OR
           -bnf (default) use Boolean Normal Form AND
           -fussy (default) use fussy encoding OR
           -sloppy for sloppy encoding (default is fussy encoding)
                 (-sloppy option can only be used with NNF formulas!)
           -gba for CGH's standard state-based symbolic automaton (default)
           -tgba for transition-based symbolic automaton instead of default state-based one
           -mcs for MCS variable order
                 -max to start with the maximum output degree node (default)
                 -min to start with the minimum output degree node
                 -random to start with a random node
                 -zero to start with the first node encountered
           -lexm for LEXM variable order
           -lexp for LEXP variable order
           -linear for Linear variable order
           -reverse to reverse a flag-specified variable ordering
                 (default is to use the Model Checker's default var order
           -v for verbose mode
```

B.2   PANDA Perl Script: Tying it all together

To enable LTL satisfiability checking and bridge the gap between the core PANDA tool for encoding LTL formulas as symbolic automata and the PBS launch script which launches multiple encodings in parallel, we use Perl, of course. In the experiments described in this thesis, we used perl scripts for generating the benchmark formulas, launching the PANDA tool, model checking the symbolic automata produced with a back-end model checker (e.g. CadenceSMV), and collecting data. A generic version of the perl code to accomplish these tasks is available online.

B.3   PANDA PBS Run Script

PANDA can be used to create several encodings of a formula, utilizing a symbolic model checker to check them for satisfiability in parallel and terminating when the first check completes, as described in Chapter 5.

We accomplish this by using the job arrays feature in Torque on the Shared University Grid at Rice (SUG@R). (The batch job scheduling system implemented on SUG@R uses the Torque package for resource management and the Maui package for job scheduling and monitoring.) Here is an example PBS job script which we tested on SUG@R; the script may need adjustments for other supercomputers:

```
#!/bin/sh

# PANDA can be run in parallel, as described in the paper
# by using the job arrays feature in Torque. Here is an example PBS job script.

## Name of the job:
#PBS -N PANDAarray

## Request wall clock time of 4 hours
#PBS -l walltime=4:00:00

## Queue name:
#PBS -q commons

## Specify number of CPUs: use 1 CPU but with exclusive access to one node
```

```
#PBS -l nodes=1:ppn=8

## Give the job exclusive access to all resources (i.e. memory) of the entire node without
##   interference from other jobs
#PBS -W x=NACCESSPOLICY:SINGLEJOB

## Merge the standard error stream with the standard output stream of the job
#PBS -j oe

## Export all environment variables to the job
#PBS -V

## Make Torque launch a job array of 30 jobs with the first PBS_ARRAYID=1
##   and the last one being 30.
#PBS -t 1-30

## This line means that you are launching the same executable 30 times but each
## instance takes a different input file, named 1.input, 2.input and so on.
ARRAY_RANGE=1-30

# Run executable: the input file contains information on which encoding to use
./PANDA.pl formula_type -r input-$PBS_ARRAYID.in

# Collect the return value of the above executable
RETVAL=$?

# Kill the array job. This will kill all the 30 instances of the script.
if [ $RETVAL -eq 0 ]; then
ARRAYID=`echo $PBS_JOBID | cut -d '[' -f1`
qdel -t $ARRAY_RANGE $ARRAYID[]
fi

#End of Sample PBS Script
```

Bibliography

[1] D. Tabakov, K. Y. Rozier, and M. Y. Vardi, "Optimized temporal monitors for SystemC," *Journal of Formal Methods in System Design*, pp. 1–33 [online], January 2012.

[2] G. Holzmann, D. Peled, and M. Yannakakis, "On nested depth-first search," in *Proceedings of the 2nd International SPIN Workshop on Model Checking of Software*, pp. 23–32, American Math. Soc., 1996.

[3] H. R. Andersen, "An Introduction to Binary Decision Diagrams." Online, September 1996.

[4] R. Bryant, "Graph-based algorithms for Boolean-function manipulation," *IEEE Transactions on Computers (TC)*, vol. C-35, no. 8, pp. 677–691, 1986.

[5] Y. Kesten, A. Pnueli, and L.-o. Raviv, "Algorithmic verification of Linear Temporal Logic specifications," in *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)* (K. G. Larsen, S. Skyum, and G. Winskel, eds.), vol. 1443 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–16, Springer-Verlag, Jul 1998.

[6] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings of the Workshop on Logic of Programs*, vol. 131 of *Lecture Notes in Computer Science (LNCS)*, pp. 52–71, Springer, 1981.

[7] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in Cesar," in *Proceedings of the 5th International Symposium on Programming*, vol. 137 of *Lecture Notes in Computer Science (LNCS)*, pp. 337–351, Springer, 1982.

[8] H. Foster, *Applied Assertion-Based Verification: An Industry Perspective*, vol. 3 of *Foundations and Trends in Electronic Design Automation*. Now Publishers, 2009.

[9] H. Foster, A. Krolnik, and D. Lacey, *Assertion-based design (2. ed.)*. Kluwer, 2004.

[10] A. Habibi and S. Tahar, "Design for verification of SystemC transaction level models," in *Design, Automation and Test in Europe*, pp. 560–565, IEEE, 2005.

[11] S. Ruah, A. Fedeli, C. Eisner, and M. Moulin, "Property-driven specification of VLSI design," tech. rep., PROSYD deliverable 1.1/1, 2005.

[12] M. Roveri, "Novel techniques for property assurance," tech. rep., PROSYD deliverable 1.2/2, 2004.

[13] K. L. McMillan, *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, CMU, Pittsburgh, PA, USA, 1992.

[14] K. McMillan, "The SMV language," tech. rep., Cadence Berkeley Lab, 1999.

[15] R. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese, "Model checking large software specifications," *IEEE Transactions on Software Engineering (TSE)*, vol. 24, pp. 156–166, 1996.

[16] T. Sreemani and J. M. Atlee, "Feasibility of model checking software requirements: A case study," in *COMPASS*, pp. 77–88, IEEE, 1996.

[17] C. Muñoz, V. Carreño, and G. Dowek, "Formal analysis of the operational concept for the Small Aircraft Transportation System," in *Rigorous Engineering of Fault-*

Tolerant Systems, vol. 4157 of *Lecture Notes in Computer Science (LNCS)*, pp. 306–325, 2006.

[18] A. Betin Can, T. Bultan, M. Lindvall, B. Lux, and S. Topp, "Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers," *Automated Software Engineering*, vol. 14, no. 2, pp. 129–178, 2007.

[19] S. P. Miller, M. W. Whalen, and D. D. Cofer, "Software model checking takes off," *Communications of the ACM*, vol. 53, no. 2, pp. 58–64, 2010.

[20] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 410–425, 2000.

[21] M. Whalen, M. Innis, J., S., and L. Wagner, "Adgs-2100 adaptive display & guidance system window manager analysis. nasa contractor report cr-2006-213952." Online, February 2006. http://shemesh.larc.nasa.gov/fm/fm-collins-pubs.html/.

[22] S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl, "Formal verification of flight critical software," *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, August 2005.

[23] M. W. Whalen, D. D. Cofer, S. P. Miller, B. H. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," in *FMICS*, pp. 68–84, 2007.

[24] P. Technology, "Prover plug-in product description." Online. http://www.prover.com/.

[25] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[26] Y. V. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking.," in *Proceedings of the 36th Design automation conference (DAC)*, pp. 300–305, 1999.

[27] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi, "A practical approach to coverage in model checking," in *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, vol. 2102 of *Lecture Notes in Computer Science (LNCS)*, pp. 66–78, Springer, 2001.

[28] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Transactions on Software Engineering (TSE)*, vol. 19, no. 1, pp. 3–12, 1993.

[29] P. C. Mehlitz and J. Penix, "Design for verification using design patterns to build reliable systems," in *Proceedings of the 6th Workshop on Component-Based Software Eng*, 2003.

[30] P. C. Mehlitz and J. Penix, "Design for verification with dynamic assertions," in *SEW*, pp. 285–292, 2005.

[31] T. M. Austin, "Design for verification?," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 80, 77, 2001.

[32] R. Schutten and T. Fitzpatrick, "Design for verification methodology allows silicon success." EEdesign, EEtimes. http://www.eetimes.com/story/OEG20030418S0043.

[33] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. MIT Press, 2008.

[34] D. Peled, "All from one, one for all: on model checking using representatives," in *Proceedings of the 5th International Conference on Computer Aided Verifica-*

tion (CAV) (C. Courcoubetis, ed.), vol. 697 of *Lecture Notes in Computer Science (LNCS)*, (Elounda, Greece), pp. 409–423, Springer, 1993.

[35] R. Milner, "An algebraic definition of simulation between programs," in *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pp. 481–489, British Computer Society, Sept 1971.

[36] C. Jones, "Specification and design of (parallel) programs," in *Information Processing 83: Proceedings of the IFIP 9th World Congress* (R. Mason, ed.), pp. 321–332, IFIP, North-Holland, 1983.

[37] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of the 5th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, vol. 1579 of *Lecture Notes in Computer Science (LNCS)*, Springer, 1999.

[38] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, (Los Angeles, California), pp. 238–252, ACM Press, New York, NY, 1977.

[39] M. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proceedings of the 1st Symposium on Logic in Computer Science*, (Cambridge), pp. 332–344, Jun 1986.

[40] M. Vardi, "Automata-theoretic model checking revisited," in *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, vol. 4349 of *Lecture Notes in Computer Science (LNCS)*, pp. 137–150, Springer, 2007.

[41] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," in *Proceedings of the 2nd International Conference on Computer Aided Verification (CAV)*, vol. 531 of *Lecture Notes in Computer Science (LNCS)*, (Rutgers), pp. 233–242, Springer, Jun 1990.

[42] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57, 1977.

[43] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL)*, (New Orleans), pp. 97–107, Jan 1985.

[44] M. Vardi and P. Wolper, "Reasoning about infinite computations," *Information and Computation*, vol. 115, pp. 1–37, Nov 1994.

[45] E. Emerson and C.-L. Lei, "Modalities for model checking: Branching time logic strikes back," in *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL)*, (New Orleans), pp. 84–96, Jan 1985.

[46] A. Sistla and E. Clarke, "The complexity of propositional linear temporal logic," *Journal ACM*, vol. 32, pp. 733–749, 1985.

[47] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti, "Formal analysis of hardware requirements," in *DAC*, pp. 821–826, ACM, 2006.

[48] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in ACTL formulas," *Journal of Formal Methods in System Design*, vol. 18, no. 2, pp. 141–162, 2001.

[49] O. Kupferman and M. Vardi, "Vacuity detection in temporal model checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, pp. 224–

233, Feb 2003.

[50] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Vardi, "A framework for inherent vacuity," in *Haifa Verification Conference*, vol. 5394 of *Lecture Notes in Computer Science (LNCS)*, pp. 7–22, Springer, 2008.

[51] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL)*, (Austin), pp. 179–190, Jan 1989.

[52] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems* (K. R. Apt, ed.), vol. F-13 of *NATO Advanced Summer Institutes*, pp. 477–498, New York, NY, USA: Springer-Verlag New York, Inc., 1985.

[53] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Safety*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.

[54] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli, "A decision algorithm for full propositional temporal logic," in *Proceedings of the 5th Conference on Computer Aided Verification (CAV)* (C. Courcoubeti, ed.), vol. 697 of *Lecture Notes in Computer Science (LNCS)*, pp. 97–109, Springer, 1993.

[55] S. Merz and A. Sezgin, "Emptiness of Linear Weak Alternating Automata," tech. rep., LORIA, December 2003.

[56] K. Rozier and M. Vardi, "LTL satisfiability checking," in *Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, vol. 4595 of *Lecture Notes in Computer Science (LNCS)*, pp. 149–167, Springer, 2007.

[57] M. De Wulf, L. Doyen, N. Maquet, and J. Raskin, "Antichains: Alternative algorithms for LTL satisfiability and model-checking," in *Proceedings of the 14th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, pp. 63–77, 2008.

[58] V. Goranko, A. Kyrilov, and D. Shkatov, "Tableau tool for testing satisfiability in LTL: Implementation and experimental analysis," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 262, pp. 113–125, 2010.

[59] G. Holzmann, "The model checker Spin," *IEEE Transactions on Software Engineering (TSE)*, vol. 23, pp. 279–295, May 1997. Special issue on Formal Methods in Software Practice.

[60] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," *Journal of Formal Methods in System Design*, vol. 1, pp. 275–288, 1992.

[61] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[62] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jeron, "On-the-fly verification of finite transition systems," *Journal of Formal Methods in System Design*, vol. 1, pp. 251–273, 1992.

[63] X. Thirioux, "Simple and efficient translation from LTL formulas to Büchi automata," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, pp. 145–159, 2002.

[64] J. Geldenhuys and H. Hansen, "Larger automata and less work for LTL model checking," in *Proceedings of the 13th International SPIN Workshop on Model Checking*

of Software, vol. 3925 of *Lecture Notes in Computer Science (LNCS)*, pp. 53–70, Springer, 2006.

[65] O. Grumberg and H. Veith, eds., *25 Years of Model Checking - History, Achievements, Perspectives*, vol. 5000 of *Lecture Notes in Computer Science (LNCS)*, Springer, 2008.

[66] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, T. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa, "VIS: A system for verification and synthesis," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, vol. 1102 of *Lecture Notes in Computer Science (LNCS)*, pp. 428–432, Springer, 1996.

[67] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, pp. 142–170, Jun 1992.

[68] E. M. Clarke, O. Grumberg, and K. Hamaguchi, "Another look at LTL model checking," *Journal of Formal Methods in System Design*, vol. 10, no. 1, pp. 47–71, 1997.

[69] E. Emerson and C.-L. Lei, "Efficient model checking in fragments of the propositional μ-calculus," in *Proceedings of the 1st IEEE Symposium on Logic In Computer Science (LICS)*, (Cambridge), pp. 267–278, Jun 1986.

[70] E. M. Clarke, "The birth of model checking," in *25 Years of Model Checking*, (http://dx.doi.org/10.1007/978-3-540-69850-0_1), pp. 1–26, 2008.

[71] J. R. Burch, E. M. Clarke, D. E. Long, K. L. Mcmillan, and D. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided*

Design of Integrated Circuits and Systems (TCAD), vol. 13, pp. 401–424, 1993.

[72] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso, "Planning via model checking: A decision procedure for AR," in *ECP*, pp. 130–142, 1997.

[73] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," in *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL)*, (Albuquerque, New Mexico), pp. 343–354, Jan 1992.

[74] A. Valmari, "A stubborn attack on state explosion," in *Proceedings of the 2nd International Conference on Computer Aided Verification (CAV)*, vol. 531 of *Lecture Notes in Computer Science (LNCS)*, (Rutgers), pp. 156–165, Springer, Jun 1990.

[75] P. Wolper, M. Vardi, and A. Sistla, "Reasoning about infinite computation paths," in *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, (Tucson), pp. 185–194, 1983.

[76] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)* (R. Alur and D. Peled, eds.), vol. 3114 of *Lecture Notes in Computer Science (LNCS)*, (Boston, MA), pp. 496–500, Springer, Jul 2004.

[77] K. G. Larsen, P. Petterson, and W. Yi, "UPPAAL: Status & developments," in *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, vol. 1254 of *Lecture Notes in Computer Science (LNCS)*, pp. 456–459, Springer, 1997.

[78] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1–2, pp. 110–122, 1997.

[79] R. Burstall, "Program proving as hand simulation with a little induction," in *Information Processing 74*, (Stockholm, Sweden), pp. 308–312, International Fed. for Information Processing, North-Holland, 1974.

[80] F. Kroger, "LAR: A logic of algorithmic reasoning," *Acta Informatica*, vol. 8, pp. 243–266, 1977.

[81] E. Clarke, O. Grumberg, and D. Long, "Verification tools for finite-state concurrent systems," in *Decade of Concurrency – Reflections and Perspectives (Proceedings of the REX School)* (J. de Bakker, W.-P. de Roever, and G. Rozenberg, eds.), vol. 803 of *Lecture Notes in Computer Science (LNCS)*, pp. 124–175, Springer, 1993.

[82] J. Kamp, *Tense Logic and the Theory of Order*. PhD thesis, UCLA, 1968.

[83] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, (New York, NY, USA), pp. 164–176, ACM, 1981.

[84] M. Y. Vardi, "From Church and Prior to PSL," in *Proceedings of the Workshop on 25 Years of Model Checking*, August 2006.

[85] W. Visser and H. Barringer, "Practical CTL* model checking: Should SPIN be extended?," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 350–365, 2000.

[86] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal analysis of fairness," in *Proceedings of the 7th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 163–173, Jan 1980.

[87] E. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science* (J. V. Leeuwen, ed.), vol. B, ch. 16, pp. 997–1072, Elsevier, MIT Press,

1990.

[88] M. Y. Vardi, "Automata-theoretic techniques for temporal reasoning," in *Handbook of Modal Logic* (F. W. Patrick Blackburn, Johan van Benthem, ed.), Elsevier, Nov 2006.

[89] O. Bernholtz, M. Y. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking," in *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)* (D. L. Dill, ed.), vol. 818 of *Lecture Notes in Computer Science (LNCS)*, pp. 142–155, Springer, Jun 1994.

[90] L. Lamport, ""sometimes" is sometimes "not never" - on the temporal logic of programs," in *Proceedings of the 7th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 174–185, Jan 1980.

[91] M. Vardi, "Branching vs. linear time: Final showdown," in *Proceedings of the 7th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, vol. 2031 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–22, Springer, 2001.

[92] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, pp. 244–263, Jan 1986.

[93] M. Fischer and R. Ladner, "Propositional dynamic logic of regular programs," *Journal of Computer and Systems Science*, vol. 18, pp. 194–211, 1979.

[94] E. Emerson and J. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," *Journal of Computer and System Science*, vol. 30, pp. 1–24, 1985.

[95] M. Vardi, "Linear vs. branching time: A complexity-theoretic perspective," in *Proceedings of the 13th IEEE Symposium on Logic In Computer Science (LICS)*, pp. 394–405, 1998.

[96] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. New York, NY, USA: Cambridge University Press, 2004.

[97] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoe-belen, *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[98] D. Giannakopoulou, *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, Technology, and Medicine, University of London, Jan 1999.

[99] E. Emerson and J. Halpern, "Sometimes and not never revisited: On branching versus linear time," *Journal ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[100] M. Vardi and L. Stockmeyer, "Improved upper and lower bounds for modal logics of programs," in *Proceedings of the 17th ACM Symposium on Theory of Computing (STOC)*, pp. 240–251, 1985.

[101] E. Emerson and C. Jutla, "The complexity of tree automata and logics of programs," in *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science (FOCS)*, (White Plains), pp. 328–337, Oct 1988.

[102] E. Emerson and A. P. Sistla, "Deciding branching time logic," in *Proceedings of the 16th ACM Symposium on Theory of Computing (STOC)*, (Washington), pp. 14–24, Apr 1984.

[103] P. Wolper, "Temporal logic can be more expressive," *Information and Control*, vol. 56, no. 1–2, pp. 72–99, 1983.

[104] J.-M. Couvreur, "Un point de vue symbolique sur la logique temporelle linéaire," in *Publications du LaCIM*, vol. 27 of *Actes du Colloque LaCIM 2000*, pp. 131–140, Université du Québec à Montréal, August 2000.

[105] B. Banieqbal and H. Barringer, "Temporal logic with fixed points," in *Temporal Logic in Specification* (B. Banieqbal, H. Barringer, and A. Pnueli, eds.), vol. 398 of *Lecture Notes in Computer Science (LNCS)*, pp. 62–74, Springer, 1987.

[106] M. Vardi, "A temporal fixpoint calculus," in *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL)*, (San Diego), pp. 250–259, Jan 1988.

[107] A. Sistla, M. Vardi, and P. Wolper, "The complementation problem for Büchi automata with applications to temporal logic," *Theoretical Computer Science*, vol. 49, pp. 217–237, 1987.

[108] E. Emerson and R. Trefler, "Generalized quantitative temporal reasoning: An automata theoretic approach," in *TAPSOFT, Proceedings of the*, vol. 1214 of *Lecture Notes in Computer Science (LNCS)*, pp. 189–200, Springer, 1997.

[109] L. Fix, "Fifteen years of formal property verification at intel," *This Volume*, 2007.

[110] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, and Y. Zbar, "The For-Spec temporal logic: A new temporal property-specification logic," in *Proceedings of the 8th International Conference on Tools and Algorithms For the Construction*

and Analysis of Systems (TACAS), vol. 2280 of *Lecture Notes in Computer Science (LNCS)*, pp. 296–211, Springer, 2002.

[111] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh, "The temporal logic Sugar," in *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, vol. 2102 of *Lecture Notes in Computer Science (LNCS)*, (Paris, France), pp. 363–367, Springer, Jul 2001.

[112] C. Eisner and D. Fisman, "Sugar 2.0 proposal presented to the accellera formal verification technical committee," 2002.

[113] M. Morley, "Semantics of temporal *e*," in *BANFF Higher Order Workshop* (T. F. Melham and F. Moller, eds.), University of Glasgow, Department of Computing Science Technical Report, 1999.

[114] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

[115] J. Havlicek and Y. Wolfsthal, "PSL and SVA: Two standard assertion languages addressing complimentary engineering needs," in *Proceedings of the Design Verification Conference*, Feb. 2005.

[116] M. Y. Vardi, "From monadic logic to PSL," in *Pillars of Computer Science* (A. Avron, N. Dershowitz, and A. Rabinovich, eds.), vol. 4800 of *Lecture Notes in Computer Science (LNCS)*, pp. 656–681, Springer, 2008.

[117] G. Rosu and S. Bensalem, "Allen Linear (interval) Temporal Logic - translation to LTL and monitor synthesis," in *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, pp. 263–277, 2006.

[118] J. F. Allen, "Towards a general theory of action and time," *Artificial Intelligence*, vol. 23, no. 2, pp. 123–154, 1984.

[119] O. Kupferman, "Sanity checks in formal verification," in *Proceedings of the 17th International Conference on Concurrency Theory*, vol. 4137 of *Lecture Notes in Computer Science (LNCS)*, pp. 37–51, Springer, 2006.

[120] R. Kurshan, *FormalCheck User's Manual*. Cadence Design, Inc., 1998.

[121] G. Ammons, D. Mandelin, R. Bodik, and J. Larus, "Debugging temporal specifications with concept analysis," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 182–195, 2003.

[122] R. Armon, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi, "Enhanced vacuity detection for linear temporal logic," in *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, Springer, 2003.

[123] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi, "Regular vacuity," in *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, vol. 3725 of *Lecture Notes in Computer Science (LNCS)*, pp. 191–206, Springer, 2005.

[124] A. Gurfinkel and M. Chechik, "How vacuous is vacuous?," in *Proceedings of the 10th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, vol. 2988 of *Lecture Notes in Computer Science (LNCS)*, pp. 451–466, Springer, 2004.

[125] A. Gurfinkel and M. Chechik, "Extending extended vacuity," in *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*

(FMCAD), vol. 3312 of *Lecture Notes in Computer Science (LNCS)*, pp. 306–321, Springer, 2004.

[126] K. Namjoshi, "An efficiently checkable, proof-based formulation of vacuity in model checking," in *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, vol. 3114 of *Lecture Notes in Computer Science (LNCS)*, pp. 57–69, Springer, 2004.

[127] M. Purandare and F. Somenzi, "Vacuum cleaning CTL formulae," in *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)* (E. Brinksma and K. G. Larsen, eds.), vol. 2404 of *Lecture Notes in Computer Science (LNCS)*, pp. 485–499, Springer, Jul 2002.

[128] O. Kupferman, M. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking," *Journal ACM*, vol. 47, pp. 312–360, Mar 2000.

[129] J. Büchi, "On a decision method in restricted second order arithmetic," in *Proceedings of the International Congress on Logic, Method, and Philosophy of Science 1960*, (Stanford), pp. 1–12, Stanford University Press, 1962.

[130] J.-M. Couvreur, "On-the-fly verification of linear temporal logic," in *Proceedings of the World Congress on Formal Methods*, pp. 253–271, 1999.

[131] M. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for Concurrency: Structure versus Automata* (F. Moller and G. Birtwistle, eds.), vol. 1043 of *Lecture Notes in Computer Science (LNCS)*, pp. 238–266, Springer, Berlin, 1996.

[132] A. Sistla, M. Vardi, and P. Wolper, "The complementation problem for Büchi automata with applications to temporal logic," in *Proceedings of the 10th Interna-*

tional Colloquium on Automata, Languages, and Programming (ICALP), vol. 194 of *Lecture Notes in Computer Science (LNCS)*, (Nafplion), pp. 465–474, Springer, Jul 1985.

[133] S. Safra and M. Vardi, "On ω-automata and temporal logic," in *Proceedings of the 21st ACM Symposium on Theory of Computing (STOC)*, (Seattle), pp. 127–137, May 1989.

[134] R. Gerth, D. Peled, M. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification, Testing and Verification (PSTV)* (P. Dembiski and M. Sredniawa, eds.), pp. 3–18, Chapman & Hall, Aug 1995.

[135] N. Daniele, F. Guinchiglia, and M. Vardi, "Improved automata generation for linear temporal logic," in *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, vol. 1633 of *Lecture Notes in Computer Science (LNCS)*, pp. 249–260, Springer, 1999.

[136] F. Somenzi and R. Bloem., "Efficient Büchi automata from LTL formulae," in *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, vol. 1855 of *Lecture Notes in Computer Science (LNCS)*, pp. 248–263, Springer, 2000.

[137] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, vol. 2102 of *Lecture Notes in Computer Science (LNCS)*, pp. 53–65, Springer, 2001.

[138] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of LTL formulae to Büchi automata.," in *Proceedings of the 22nd IFIP International*

Conference on Formal Techniques for Networked and Distributed Systems, pp. 308–326, 2002.

[139] R. Sebastiani and S. Tonetta, ""more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking," in *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, vol. 2860 of *Lecture Notes in Computer Science (LNCS)*, pp. 126–140, Springer, 2003.

[140] A. Duret-Lutz and D. Poitrenaud, "Spot: An extensible model checking library using transition-based generalized Büchi automata," in *Proceedings of the 12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 76–83, IEEE, 2004.

[141] C. Fritz, "Concepts of automata construction from LTL," in *LPAR, Proceedings of the 12th International Conference*, vol. 3835 of *Lecture Notes in Computer Science (LNCS)*, pp. 728–742, Springer, 2005.

[142] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 3, no. 2, pp. 125–143, 1977.

[143] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, pp. 455–495, Jul 1982.

[144] O. Kupferman and M. Vardi, "Model checking of safety properties," in *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, vol. 1633 of *Lecture Notes in Computer Science (LNCS)*, pp. 172–183, Springer, 1999.

[145] B. Alpern and F. Schneider, "Recognizing safety and liveness," *Dist. computing*, vol. 2, pp. 117–126, 1987.

[146] E. Kindler, "Safety and liveness properties: A survey," *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, vol. 53, pp. 268 – 272, Jun 1994.

[147] M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider, *Dist. systems: methods and tools for specification. An advanced course*. NY, USA: Springer, 1985.

[148] A. Sistla, "Safety, liveness, and fairness in temporal logic," *Formal Aspects of Computing*, vol. 6, pp. 495–511, 1994.

[149] B. Alpern and F. Schneider, "Defining liveness," *Information processing letters*, vol. 21, pp. 181–185, 1985.

[150] R. Kurshan, *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[151] S. Safra, "On the complexity of ω-automata," in *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science (FOCS)*, (White Plains), pp. 319–327, Oct 1988.

[152] O. Kupferman and M. Vardi, "Weak alternating automata are not that weak," in *Proceedings of the 5th Israeli Symposium on Theory of Computing and Systems (ISTCS)*, pp. 147–158, IEEE Press, 1997.

[153] L. Doyen and J.-F. Raskin, "Improved algorithms for the automata-based approach to model-checking," in *Proceedings of the 13th International Conference on Tools*

and Algorithms For the Construction and Analysis of Systems (TACAS), vol. 4424 of *Lecture Notes in Computer Science (LNCS)*, pp. 451–465, Springer, 2007.

[154] S. Fogarty and M. Y. Vardi, "Büchi complementation and size-change termination," in *Proceedings of the 15th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, (Berlin, Heidelberg), pp. 16–30, Springer, 2009.

[155] P. Linz, *An introduction to formal languages and automata (2nd ed.)*, ch. 1–4. Lexington, MA, USA: D. C. Heath and Company, 1996.

[156] P. Gribomont and P. Wolper, "Temporal logic, in from modal logic to deductive databases," *A. Thayse, Editor*, 1989.

[157] M. Michel, "Complementation is more difficult with automata on infinite words." CNET, Paris, 1988.

[158] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, 2003.

[159] C. Lee, "Representation of switching circuits by binary decision programs," in *Bell Syst. Tech. Journal*, vol. 38, pp. 985–999, Jul 1959.

[160] S. B. Akers, "Binary decision diagrams," in *IEEE Transactions on Computers (TC)*, vol. C- 27, no. 6, pp. 509–516, Jun 1978.

[161] S. Fortune, J. E. Hopcroft, and E. M. Schmidt, "The complexity of equivalence and containment for free single variable program schemes," in *Proceedings of the 5th International Colloquium on Automata, Languages, and Programming (ICALP)*, (London, UK), pp. 227–240, Springer, 1978.

[162] R. Bryant, "Symbolic Boolean manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[163] J. Esparza, "Decidability of model checking for infinite-state concurrent systems," *Acta Informatica*, vol. 34, pp. 85–107, 1997.

[164] D. Sieling, "The nonapproximability of OBDD minimization," *Information and Computation*, vol. 172, pp. 103–138, 1998.

[165] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," in *DAC, Proceedings of the 27th ACM/IEEE Conference*, (New York, NY, USA), pp. 52–57, ACM, 1990.

[166] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Markovian analysis of large finite state machines," *IEEE Transactions on CAD*, vol. 15, pp. 1479–1493, 1996.

[167] M. Chechik, S. Easterbrook, and B. Devereux, "Model checking with multi-valued temporal logics," in *ISMVL*, pp. 187–192, 2000.

[168] D. Sieling and I. Wegener, "Reduction of OBDDs in linear time," *Information Processing Letters*, vol. 48, no. 3, pp. 139–144, 1993.

[169] R. Streett, "Propositional dynamic logic of looping and converse," *Information and Control*, vol. 54, pp. 121–141, 1982.

[170] N. Francez and D. Kozen, "Generalized fair termination," in *Proceedings of the 11th Symposium on Principles of Programming Languages (POPL)*, pp. 46–53, 1984.

[171] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *Proceedings of the 32nd Design Automation Conference*, pp. 427–432, IEEE, 1995.

[172] R. Hojati, R. K. Brayton, and R. P. Kurshan, "BDD-based debugging of design using language containment and fair CTL," in *Proceedings of the 5th International Conference on Computer Aided Verification (CAV)*, (London, UK), pp. 41–58, Springer-Verlag, 1993.

[173] K. Ravi, R. Bloem, and F. Somenzi, "A comparative study of symbolic algorithms for the computation of fair cycles," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, no. 1954 in Lecture Notes in Computer Science (LNCS), pp. 143–160, Springer, 2000.

[174] R. Hojati, H. Touati, R. Kurshan, and R. Brayton, "Efficient ω-regular language containment," in *Proceedings of the 4th International Conference on Computer Aided Verification (CAV)*, vol. 663 of *Lecture Notes in Computer Science (LNCS)*, Springer, 1992.

[175] R. Hardin, R. Kurshan, S. Shukla, and M. Vardi, "A new heuristic for bad cycle detection using BDDs," in *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, vol. 1254 of *Lecture Notes in Computer Science (LNCS)*, pp. 268–278, Springer, 1997.

[176] A. Xie and P. Beerel, "Implicit enumeration of strongly connected components," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, (San Jose, CA, USA), pp. 37–40, ACM, IEEE Press, Nov 1999.

[177] A. Xie and P. A. Beerel, "Implicit enumeration of strongly connected components and an application to formal verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 19, no. 10, pp. 1225–1230, 2000.

[178] R. Bloem, H. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in $n \log n$ symbolic steps," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, vol. 1954 of *Lecture Notes in Computer Science (LNCS)*, pp. 37–54, Springer, 2000.

[179] K. Fisler, R. Fraer, G. Kamhi, M. Vardi, and Z. Yang, "Is there a best symbolic cycle-detection algorithm?," in *Proceedings of the 7th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, vol. 2031 of *Lecture Notes in Computer Science (LNCS)*, pp. 420–434, Springer, 2001.

[180] S. Ben-David, J. Pound, R. J. Trefler, D. Tsarkov, and G. E. Weddell, "Fair cycle detection using description logic reasoning," in *Description Logics*, 2009.

[181] A. C. R. Cavada, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, "NuSMV 2.4 user manual," tech. rep., CMU and ITC-irst, 2005.

[182] Cadence, "SMV." http://www.cadence.com/company/cadence_labs_research.html.

[183] P. Wolper, "An introduction to model checking," in *Proceedings of the Software Quality Week (SQW'95)*, (San Francisco, CA), May 1995.

[184] J. Hooker, "Testing heuristics: We have it all wrong," *Journal of Heuristics*, vol. 1, pp. 33–42, 1995.

[185] H. Tauriainen and K. Heljanko, "Testing LTL formula translation into Büchi automata," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, no. 1, pp. 57–70, 2002.

[186] K. Etessami and G. Holzmann, "Optimizing Büchi automata.," in *Proceedings of the 11th International Conference on Concurrency Theory*, vol. 1877 of *Lecture Notes in Computer Science (LNCS)*, pp. 153–167, Springer, 2000.

[187] J. Geldenhuys and A. Valmari, "Tarjan's algorithm makes on-the-fly LTL verification more efficient," in *Proceedings of the 10th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, vol. 2988 of *Lecture Notes in Computer Science (LNCS)*, pp. 205–219, Springer, 2004.

[188] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[189] T. Latvala, "Efficient model checking of safety properties," in *Proceedings of the 10th International SPIN Workshop on Model Checking of Software*, vol. 2648 of *Lecture Notes in Computer Science (LNCS)*, pp. 74–88, 2003.

[190] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, 1981.

[191] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[192] R. Pelánek, "BEEM: benchmarks for explicit model checkers," in *Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, vol. 4595 of *Lecture Notes in Computer Science (LNCS)*, (Berlin, Heidelberg), pp. 263–267, Springer-Verlag, 2007.

[193] M. Kamel and S. Leue, "Validation of a remote object invocation and object migration in CORBA GIOP using Promela/Spin," in *Proceedings of the 4th International SPIN Workshop on Model Checking of Software*, 1998.

[194] Y. Dong, X. Du, G. J. Holzmann, and S. A. Smolka, "Fighting livelock in the GNU i-protocol: a case study in explicit-state model checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, no. 4, pp. 505–528, 2003.

[195] R. Kaivola, *Using automata to characterise fixed-point temporal logics*. PhD thesis, University of Edinburgh, 1997.

[196] C. Fritz, "Constructing Büchi automata from linear temporal logic using simulation relations for alternating bchi automata," in *Proceedings of the 8th International Conference on Implementation and Application of Automata*, no. 2759 in Lecture Notes in Computer Science (LNCS), pp. 35–48, Springer, 2003.

[197] G. Pan, U. Sattler, and M. Vardi, "BDD-based decision procedures for K," in *Proceedings of the 18th International Conference on Automated Deduction*, vol. 2392 of *Lecture Notes in Computer Science (LNCS)*, pp. 16–30, Springer, 2002.

[198] N. Piterman and M. Vardi, "From bidirectionality to alternation," *Theoretical Computer Science*, vol. 295, pp. 295–321, Feb 2003.

[199] R. Sebastiani, S. Tonetta, and M. Vardi, "Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking," in *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, vol. 3576 of *Lecture Notes in Computer Science (LNCS)*, pp. 350–373, Springer, 2005.

[200] M. Vardi, "Nontraditional applications of automata theory," in *Proceedings of the International Symposium on Theoretical Aspects of Computer Software*, vol. 789 of *Lecture Notes in Computer Science (LNCS)*, pp. 575–597, Springer, 1994.

[201] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[202] R. Bloem, K. Ravi, and F. Somenzi, "Efficient decision procedures for model checking of linear time logic properties," in *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, vol. 1633 of *Lecture Notes in Computer Science (LNCS)*, pp. 222–235, Springer, 1999.

[203] S. Bensalem, V. Ganesh, Y. Lakhnech, C. M. noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari, "An overview of SAL," in *LFM 2000: Fifth NASA Langley Formal Methods Workshop* (C. M. Holloway, ed.), (Hampton, VA), pp. 187–196, NASA Langley Research Center, Jun 2000.

[204] A. Cimatti, M. Roveri, and S. Tonetta, "Syntactic optimizations for PSL verification," in *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, Proceedings of the 13th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS), (Berlin, Heidelberg), pp. 505–518, Springer-Verlag, 2007.

[205] A. Ferrara, G. Pan, and M. Y. Vardi, "Treewidth in verification: Local vs. global," in *LPAR, Proceedings of the 12th International Conference*, vol. 3835 of *Lecture Notes in Computer Science (LNCS)*, pp. 489–503, Springer, 2005.

[206] J. Cichon, A. Czubak, and A. Jasinski, "Minimal Büchi automata for certain classes of LTL formulas," *Dependability of Computer Systems*, vol. 0, pp. 17–24, 2009.

[207] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment," in *IFIG Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, vol. 3725 of *Lecture Notes in Computer Science (LNCS)*, pp. 254–268, Springer, 2005.

[208] P. Arcaini, A. Gargantini, and E. Riccobene, "Automatic review of abstract state machines by meta property verification," in *NFM, NASA/CP-2010-216215* (C. Muñoz, ed.), (Langley Research Center, Hampton VA 23681-2199, USA), pp. 4–13, NASA, April 2010.

[209] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel, "Treewidth: Computational experiments," ZIB-Report 01–38, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Berlin, Germany, 2001. Also available as technical report UU-CS-2001-49 (Utrecht University) and research memorandum 02/001 (Universiteit Maastricht).

[210] L. Pulina and A. Tacchella, "A self-adaptive multi-engine solver for quantified Boolean formulas," *Constraints 14*, vol. 14, no. 1, pp. 80–116, 2009.

[211] E. Filiot, N. Jin, and J.-F. Raskin, "An antichain algorithm for LTL realizability," in *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pp. 263–277, 2009.

[212] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Automatic hardware synthesis from specifications: A case study," in *DATE*, pp. 1188–1193, 2007.

[213] K. Schneider, "Improving automata generation for linear temporal logic by considering the automaton hierarchy," in *LPAR*, (London, UK), pp. 39–54, Springer-Verlag, 2001.

[214] R. Bloem, A. Cimatti, I. Pill, and M. Roveri, "Symbolic implementation of alternating automata," *International Journal Foundations Computer Science*, vol. 18, no. 4, pp. 727–743, 2007.

[215] M. Fisher, "A normal form for temporal logics and its applications in theorem-proving and execution," *Journal Log. Comput.*, vol. 7, no. 4, pp. 429–456, 1997.

[216] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," in *Proceedings of the 7th International Workshop on Formal Methods for Industrial*

Critical Sys., vol. 66:2 of *Electronic Notes in Theoretical Computer Science*, 2002.

[217] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta, "From psl to nba: A modular symbolic encoding," in *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2006.

[218] M. Järvisalo, "Structure-based satisfiability checking: Analyzing and harnessing the potential," *Artificial Intelligence Commun.*, vol. 22, pp. 117–119, Apr. 2009.

[219] M. Gagliolo and J. Schmidhuber, "Learning dynamic algorithm portfolios," *Annals of Mathematics and Artificial Intelligence*, vol. 47, pp. 295–328, Aug 2006.

[220] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown, "Performance prediction and automated tuning of randomized and parametric algorithms," in *Proceedings of the 12th international conference on Principles and Practice of Constraint Programming*, CP'06, (Berlin, Heidelberg), pp. 213–228, Springer-Verlag, 2006.

[221] O. Trachsel and T. R. Gross, "Variant-based competitive parallel execution of sequential programs," in *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, (New York, NY, USA), pp. 197–206, ACM, 2010.

[222] K. Havelund and G. Rosu, "Synthesizing monitors for safety properties," in *Proceedings of the 8th International Conference on Tools and Algorithms For the Construction and Analysis of Systems (TACAS)*, vol. 2280 of *Lecture Notes in Computer Science (LNCS)*, pp. 342–356, Springer, 2002.

[223] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout, "Reasoning with temporal logic on truncated paths," in *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, vol. 2725 of *Lecture Notes in Computer Science (LNCS)*, pp. 27–39, Springer, 2003.

[224] R. Armoni, D. Korchemny, A. Tiemeyer, and M. V. Y. Zbar, "Deterministic dynamic monitors for linear-time assertions," in *Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification*, vol. 4262 of *Lecture Notes in Computer Science (LNCS)*, Springer, 2006.

[225] M. d'Amorim and G. Roșu, "Efficient monitoring of ω-languages," in *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, vol. 3576 of *Lecture Notes in Computer Science (LNCS)*, pp. 364–378, Springer, 2005.

[226] L. J. Jagadeesan, C. Puchol, and J. E. V. Olnhausen, "Safety property verification of esterel programs and applications to telecommunications software," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, vol. 939 of *Lecture Notes in Computer Science (LNCS)*, pp. 127–140, Springer, 1996.

[227] K. Schneider and D. W. Hoffmann, "A hol conversion for translating linear time temporal logic to omega-automata.," in *TPHOLs*, pp. 255–272, 1999.

[228] K. Rozier and M. Vardi, "LTL satisfiability checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, pp. 123 – 137, March 2010.

[229] R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M. Vardi, "Efficient LTL compilation for SAT-based model checking," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 877–884, 2005.

[230] O. Kupferman and M. Vardi, "Model checking of safety properties," *Journal of Formal methods in System Design*, vol. 19, pp. 291–314, Nov 2001.

[231] A. Møller, "dk.brics.automaton." http://www.brics.dk/automaton/, 2004.

[232] M. D. Wulf, L. Doyen, N. Maquet, and J.-F. Raskin, "Alaska," in *Automated Technology for Verification and Analysis (ATVA)*, pp. 240–245, 2008.

[233] N. Maquet, *New Algorithms and Data Structures for the Emptiness Problem of Alternating Automata*. PhD thesis, Université Libre de Bruxelles, 2011.

[234] V. Schuppan and L. Darmawan, "Evaluating LTL satisfiability solvers," in *Proceedings of the 9th international conference on Automated technology for verification and analysis*, Automated Technology for Verification and Analysis (ATVA), (Berlin, Heidelberg), pp. 397–413, Springer-Verlag, 2011.

[235] K. Klai and D. Poitrenaud, "MC-SOG: An LTL model checker based on symbolic observation graphs," in *Proceedings of the 29th international conference on Applications and Theory of Petri Nets*, PETRI NETS '08, (Berlin, Heidelberg), pp. 288–306, Springer-Verlag, 2008.

[236] R. Ehlers and B. Finkbeiner, *On the virtue of patience: minimizing Büchi automata.*, pp. 129–145. Enschede, The Netherlands: Berlin: Springer, September 2010.

[237] A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg, "Combining explicit and symbolic approaches for better on-the-fly LTL model checking," *CoRR*, vol. abs/1106.5700, 2011.

[238] T. Babiak, M. Kretínský, V. Rehák, and J. Strejček, "LTL to Büchi automata translation: Fast and more deterministic," *CoRR*, vol. abs/1201.0682, 2012.

[239] T. Babiak, "Translation of LTL to ω-automata [online]." Masarykova univerzita, Fakulta informatiky [online], 2012 [cit. 2012-06-13]. Dostupn z WWW ¡http://is.muni.cz/th/143254/fi_r/¿.

[240] J. Strejček, "From infinite-state systems to translation of LTL to automata." Habilitation Thesis (Collection of Articles), Masaryk University, April 2012.

[241] M. Montali, P. Torroni, M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, and P. Mello, "Verification from declarative specifications using logic programming," in *Proceedings of the 24th International Conference on Logic Programming*, ICLP, (Berlin, Heidelberg), pp. 440–454, Springer-Verlag, 2008.

[242] G. Geeraerts, G. Kalyon, T. Le Gall, N. Maquet, and J.-F. Raskin, "Lattice-valued binary decision diagrams," in *Proceedings of the 8th international conference on Automated technology for verification and analysis*, Automated Technology for Verification and Analysis (ATVA), (Berlin, Heidelberg), pp. 158–172, Springer-Verlag, 2010.

[243] M. Pradella, A. Morzenti, and P. San Pietro, "Refining real-time system specifications through bounded model- and satisfiability-checking," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, (Washington, DC, USA), pp. 119–127, IEEE Computer Society, 2008.

[244] M. Pradella, A. Morzenti, and P. S. Pietro, "A metric encoding for bounded model checking (extended version)," *CoRR*, vol. abs/0907.3085, 2009.

[245] Y. Zhao and K. Y. Rozier, "Formal specification and verification of a coordination protocol for an automated air traffic control system," *under submission*, p. TBD, 2012.

[246] S. Ben-David, M. Chechik, A. Gurfinkel, and S. Uchitel, "CSSL: a logic for specifying conditional scenarios," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE, (New York, NY, USA), pp. 37–47, ACM, 2011.