Symbolic Model Checking for Large Software Specifications

William Chan

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science and Engineering
University of Washington, 1999
© Copyright 1999 William Chan

Extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to the author.
Abstract

Model checking is the most successful formal technique for verifying hardware behaviors. A model checker exhaustively explores a state machine's state space to check it against a temporal-logic formula. The technique is therefore restricted by the huge number of states that many real-life systems possess. Symbolic model checking overcomes this fundamental limitation by representing sets of states implicitly as binary decision diagrams (BDDs), which can often succinctly capture the regularity in many industrial circuits.

Despite its tremendous success in hardware, BDD-based model checking has rarely been applied to software. The prevalent view postulates that BDDs cannot capture the complexity inherent in software systems. Contrary to this belief, we present some results and experience in applying the technique to the state-machine specifications of two software systems: a collision-avoidance system and an electrical power distribution system, both used on commercial aircraft.

The two models are written in variants of the statecharts language, a popular specification language for reactive software. We systematically translated portions of the models to inputs to a BDD-based model checker. The model-checking analyses disclosed subtle but important errors not found in prior verification efforts. Despite the final encouraging results, the huge BDDs generated made many of the initial analyses infeasible. We have developed intuition about some reasons for BDD blowups, and based on the insights, modified the models and the model checker to speed up the analyses by orders of magnitude. These optimizations exploit the models' synchronization mechanisms and environmental assumptions, and enable many analyses that were previously intractable.

Another limitation of BDDs is their exponential complexity for nonlinear arithmetic. This poses a problem for verifying embedded systems that base their transitions on nonlinear predicates over sensor inputs. To attack the problem, we have extended the conventional model checking by tightly coupling BDDs with a nonlinear constraint solver. This extension and the optimizations above have broadened the range of systems that are amenable to automatic behavioral verification, and are important steps toward making symbolic model checking a viable verification technique for state-machine specifications.
Table of Contents

List of Figures v

List of Tables vii

Biography ix

Acknowledgments xi

Chapter 1: Introduction 1
  1.1 Background ......................................................... 2
  1.2 Contributions ..................................................... 4
  1.3 Related Work ....................................................... 5
  1.4 Organization ....................................................... 6

Chapter 2: Symbolic Model Checking 7
  2.1 Model Checking .................................................... 7
  2.2 Binary Decision Diagrams ........................................ 8
  2.3 SMV ............................................................... 10

Chapter 3: Statecharts and RSML 13
  3.1 Syntax and Semantics ............................................. 13
  3.2 Translation Example ............................................. 17
  3.3 Translation Rules ................................................ 20
    3.3.1 Statecharts as Transition Systems ....................... 20
    3.3.2 Translating Global States ................................ 23
    3.3.3 Translating Deterministic Transitions .................. 24
    3.3.4 Translating Nondeterministic Transitions ............... 26
    3.3.5 Translating Timing Constraints ........................... 28
    3.3.6 Translating Previous Values ............................... 29
    3.3.7 Miscellaneous ................................................ 30
  3.4 Discussion and Related Work .................................... 31
Chapter 4: Case Study: TCAS II

4.1 General Description .................................................. 33
4.2 Obstacles to Model Checking ........................................ 34
  4.2.1 TCAS II .......................................................... 34
  4.2.2 BDDs ............................................................ 36
  4.2.3 SMV .............................................................. 36
4.3 Results ................................................................. 38
  4.3.1 Transition Consistency ......................................... 38
  4.3.2 Function Consistency .......................................... 40
  4.3.3 Macrostep Termination ....................................... 42
  4.3.4 Inhibition of Resolution Advisories ......................... 42
  4.3.5 Output Agreement .............................................. 42
  4.3.6 Mutual Exclusion of Corrective Climb and Descent .......... 43
  4.3.7 Miscellaneous .................................................. 44
4.4 Discussion ........................................................... 44
  4.4.1 Related Work ................................................ 44
  4.4.2 Model Checking as a Design Tool ............................ 46

Chapter 5: Case Study: EPD System

5.1 General Description .................................................. 49
5.2 Obstacles to Model Checking ........................................ 50
5.3 Results ................................................................. 50
5.4 Discussion ........................................................... 52

Chapter 6: State-Set Optimizations

6.1 Pruning Backward Traversals ...................................... 55
  6.1.1 An Example with Mutually Exclusive Events ................. 56
  6.1.2 Pruning Using Invariants ..................................... 57
  6.1.3 Experimental Results ........................................ 58
6.2 Oblivious vs. Non-Oblivious Synchronization ..................... 58
  6.2.1 Difference in Efficiency of Analyses ....................... 59
  6.2.2 Experimental Results ........................................ 62
6.3 Microstep Counter .................................................. 62
  6.3.1 Construction of the Microstep Counter ...................... 63
  6.3.2 Benefits of the Microstep Counter .......................... 65
  6.3.3 Experimental Results ........................................ 66
  6.3.4 Condition-Driven Transitions ............................... 66
6.4 Forward vs. Backward Traversals .................................. 68
  6.4.1 Performance Difference Between Forward and Backward Traversals . 68
  6.4.2 Implications on Counterexample Search ..................... 69
6.5 Discussion and Related Work ...................................... 70
Chapter 7: Other Optimizations

7.1 Partitioning Transition Relations ........................................ 73
  7.1.1 Background ...................................................... 73
  7.1.2 Determining a Conjunctive Partition .............................. 74
  7.1.3 Disjunctive Partitioning ......................................... 74
  7.1.4 DNF Partitioning and Serialization ............................... 75
  7.1.5 Experimental Results .......................................... 77
7.2 Automatic Abstraction ..................................................... 79
  7.2.1 Dependency Analysis .............................................. 79
  7.2.2 Experimental Results .......................................... 80
7.3 Short-Circuiting .......................................................... 82
7.4 Discussion and Related Work ............................................ 82

Chapter 8: Nonlinear Arithmetic ................................................. 85

8.1 Overview .................................................................. 85
8.2 System Models ............................................................ 86
  8.2.1 Labeled Transition Systems ....................................... 87
  8.2.2 Constrained Transition Systems ................................. 87
8.3 Model Checking for Constrained Transition Systems ............... 89
  8.3.1 Boolean Encoding of the Quotient System ..................... 89
  8.3.2 Filtering BDDs ...................................................... 90
  8.3.3 A Filtering Algorithm ............................................ 93
8.4 Example .................................................................. 94
8.5 Discussion and Related Work .......................................... 96

Chapter 9: Lessons Learned and Future Work .................................. 99

Appendix A: Program Listings ...................................................... 103
  A.1 SMV Program for the Translation Example ....................... 103
  A.2 SMV Programs for the Experiments on the Parameterized Examples 105

Bibliography ............................................................................. 109
List of Figures

2.1 Algorithm for computing $Pr^*(E)$ ............................................ 8
2.2 BDD for odd parity ............................................................... 9
3.1 An example of a statecharts machine ........................................ 14
3.2 State hierarchy drawn as tree .................................................. 15
3.3 Microstep, macrostep, and synchrony hypothesis ............................. 16
3.4 Transition from Off to On ...................................................... 17
3.5 Nested or-states in TCAS II .................................................... 24
3.6 Rules for declaring and initializing SMV variables for statecharts ......... 25
3.7 Rules for translating deterministic statecharts transitions .................. 25
3.8 Rules for translating a class of nondeterministic statecharts transitions 27
3.9 Transitions with same sources and destinations .............................. 27
4.1 Own-Aircraft in TCAS II SRS .................................................. 34
4.2 Advisory-Status in TCAS II SRS .............................................. 35
4.3 One of the state transitions in Own-Aircraft .................................. 35
4.4 Definition of Displayed-Model-Goal ......................................... 41
5.1 Handling a power-source failure in the EPD model ........................... 49
5.2 A circuit breaker and its controller .......................................... 50
6.1 State machine with mutually exclusive events and non-oblivious synchronization 56
6.2 Oblivious Synchronization ..................................................... 59
6.3 The extended precedence relation $\prec$ ....................................... 60
6.4 Backward search for the non-oblivious model ................................ 61
6.5 Backward search for the oblivious model .................................... 61
6.6 Effects of microstep counter on system behaviors ............................ 64
6.7 Transitions guarded with microstep counter .................................. 65
6.8 Event sequences of the non-oblivious system with microstep counter .... 65
6.9 Original Algorithm for counterexample search ................................ 69
6.10 Modified algorithm for counterexample search ............................... 70
7.1 One event triggering two state machines .................................... 76
7.2 The serialized machine .......................................................... 76
7.3 False dependency ................................................. 81
8.1 A BDD filtering algorithm ........................................ 93
8.2 A hypothetical automobile cruise-control system with collision avoidance ... 95
List of Tables

6.1 Performance of pruning for TCAS II ................................. 58
6.2 Performance for the parameterized examples .................... 62
6.3 Performance of using microstep counter for the EPD model .... 66

7.1 Performance of different ways of partitioning TCAS II .......... 78
7.2 Performance for the abstract models ............................... 81
7.3 Performance of short-circuiting ..................................... 82
Biography

William Chan was born in December 1972 in Hong Kong. He went to Queen’s College there for secondary school, and in 1991 moved to the United States for undergraduate education. In 1994, he received his Bachelor of Science with Distinction from the Department of Computer Science at Cornell University in Ithaca, New York. He joined the Department of Computer Science and Engineering at the University of Washington in Seattle in the same year, and obtained his Master of Science in 1996. In his spare time, he likes to dream of having a career as a nature photographer.
Acknowledgments

This research would not have been possible without the insights and guidance of Richard Anderson, Paul Beame, and David Notkin. They gave me tremendous amount of freedom in the pursuit of the ideas in this dissertation, and steered me away from the more dubious ones. Some results in this dissertation are consequences of their keen observations.

Fellow students in the CSE 590MC and CSE 590N seminars at the University of Washington, notably Mike Ernst, Todd Millstein, Kurt Partridge, Sean Sandys, and Jayram Thathachar, helped me understand the technical details and shaped my view on the subject. Richard Ladner explained to me some theoretical aspects of temporal logics. In the TCAS II case study, Jon Reese taught me the intricacies of RSML and the TCAS II requirements, while Steve Burns helped me with the symbolic model checker when I still had no clue what it does. Bill Warner and Dave Jones kindly provided their statecharts model of the electrical power distribution system, and Greg Taleck's initial investigation of the model facilitated the study.

I would also like to thank the speakers and other participants in the 1997 NATO-ASI International Summer School on Verification of Digital and Hybrid Systems in Antalya, Turkey. The discussions with Tevfik Bultan during and after the summer school were particularly fruitful.

My heartfelt thanks go to Lady Youde and the Sir Edward Youde Memorial Fund Council in Hong Kong for their generous financial support for my undergraduate education. I would have taken a very different career route without them.

The work in this dissertation was supported in part by a Microsoft-endowed graduate fellowship and National Science Foundation grants CCR-9706070 and CCR-9700660.

Last but not least, I'm indebted to Vivian, whose unfailing love, understanding, and passion throughout the years transcended the physical distance.
Introduction

Errors in software specifications are costly and, in some cases, threaten lives [16, 106]. How can we increase our confidence in the specifications, particularly those of safety-critical systems? Inspection, simulation, and testing are useful, but provide a low degree of guarantee and are time-consuming. Formal methods offer opportunities for mechanical verification, but most existing techniques either do not scale to large systems, require extensive human efforts, or are limited to verifying simple, though important, properties like deadlock freedom, consistency, and completeness.

Symbolic model checking [28] based on binary decision diagrams (BDDs) [22] is an efficient automatic verification technique that is simultaneously capable of scaling and of verifying a wide range of properties. It has been applied successfully to many industry-scale hardware circuits, but not aggressively to the analysis of software specifications. Indeed, it was widely believed that the technique is not effective for software systems, as suggested by the following remarks from well-known software and verification researchers.

“[BDD-based model checking] works well for hardware designs with regular logical structures. For such systems, BDD representations can reduce the state space from exponential order of the number of state variables to linear. However, it is less likely to achieve similar reductions in software specifications whose logical structures are less regular.” [39, p. 51]

“For reasons not well understood, BDDs are often able to exploit the regularity that is readily apparent even to the human eye in many hardware designs. Because software typically lacks this regularity, BDD-based model checking seems much less helpful for software verification.” [56, p. 206].

“The time and space complexity of the symbolic approach is affected not only by the size of the specification but also by the regularity of specification. Software requirements specifications lack this necessary regular structure, and it is unclear how well the symbolic approach will perform on these specifications.” [77, p. 364]

This dissertation sets out to refute such beliefs.
1.1 Background

The major focus of this work is on verifying specifications of reactive safety-critical software. The emphasis on safety-critical systems can be easily understood: These are the systems whose behaviors can directly determine life and death. Examples include the software used in aircraft, railroad crossings, automobiles, nuclear power plants, etc. In contrast to transformational systems, which compute a function and terminate, these systems are reactive in that they maintain a non-terminating interaction with the environment, which include inputs from human operators and from the physical process being controlled.

In this domain, the most critical and expensive errors occur not in the implementation, but rather in the high-level specification from which the code is derived [106]. Bugs in the code can certainly be hazardous as well, but they tend to be easier to fix. Specification flaws, however, can have far-reaching consequences, and fixing them may require redesigning a significant portion of the system. Hence, identifying problems in the specification early is of utmost importance.

Software specifications written in natural languages are almost always vague and ambiguous and impossible to analyze rigorously with current technology. Therefore, formal specification languages with precise syntax and semantics are needed. (Often, the process of writing the specification precisely alone can expose a number of conceptual flaws.) Such languages come in many varieties [69, 84, 103, 113, 130]. Many of them, however, are ill-suited for describing the kind of reactive process-control systems that we are interested in. Furthermore, they are often based on traditional mathematical notations, which many domain experts without formal training in computer science find unnatural.

For our purposes, the most popular formalisms are based on state machines, whose intuitive notations and mental models are favored by application experts such as aircraft engineers. These languages include the statecharts language (particularly the one implemented in the STATEMATE toolset [72]), one of its variants the Requirements State Machine Language (RSML) [107], and the notation of Software Cost Reduction (SCR) based on Parnas's tables [81]. Though different in syntax, semantics, and assumptions made on the environment, all of these formalisms represent the embedded controller that we try to build as interacting synchronous state machines. Each state machine models the behaviors of a component, and together they specify an unambiguous mapping from environmental inputs to actuator outputs. The inputs may come from human operators or from sensors that measure quantities such as the altitude of the aircraft or the temperature of a nuclear reactor. The outputs may be displayed to the human operators, or directly used to control the physical process.

Some successful industrial applications of these state-machine specification languages include the official RSML system requirements specification of the Traffic Alert and Collision Avoidance System II (TCAS II), an airborne collision avoidance system used on most commercial aircraft in the United States [124], and the SCR-based requirements of U.S. Navy's A-7 aircraft [1] and Lockheed's C-130J aircraft [59]. The statecharts language has even been adopted by the Unified Modeling Language (UML) proposed by the industry to model object behaviors [18].
Although formal and in principle suitable for mechanical analysis, the verification and validation of specifications written in these languages are usually limited to informal techniques like inspection and simulation, and local techniques like consistency and completeness checking [77, 79]. Techniques like theorem proving are too laborious to apply to most real systems. In sharp contrast, for many hardware designs whose errors can cost the manufacturer millions of dollars, symbolic model checking, which provides much stronger correctness guarantee than informal or local techniques, has become the preferred verification method by practitioners.

Simply put, model checking [40, 123] is an extreme form of simulation or testing: Every possible behavior of the system is checked for correctness. The system is modeled as a finite-state machine (which may be the product of a collection of concurrent state machines) and the criterion for correctness is specified as a formula in a temporal logic [120]. Temporal logics can express a wide range of behavioral properties; for example, a set of error states is not reachable; a request is always followed by a response; a response cannot precede a request; the system is stable infinitely often; etc. The state space of the model is then exhaustively explored by depth-first or breadth-first search to either establish its correctness or locate a faulty behavior. Though conceptually simple, this explicit state-enumeration approach seriously suffers from the state-explosion problem—the number of states grows exponentially in the number of concurrent components, and even a simple design can have an astronomical number of states.

A breakthrough came with the advent of symbolic search techniques, which circumvent this bottleneck by visiting a whole set of states rather than an individual state at a time [49, 111]. A state set is encoded as a Boolean function, which in turn represented by a graphical data structure called a BDD. The state-space exploration is then reduced to manipulating these data structures, and the sheer number of states is no longer the obstacle to analysis. Instead, the limitation is the size of the BDDs, which depends on the structure of the model. Quoting from Hu [90]:

"The number of states in a system, the number of latches or variables, the number of gates, the number of lines of code, and other standard measures of the complexity of a system are all poor measures of how difficult the system is to verify with BDDs. The real issue is BDD size, and BDD size doesn’t correlate well with any of these measures. Instead, BDD size captures some notion of the communication complexity in the design.”

For this reason, predicting and therefore improving the efficiency of BDD-based algorithms are notoriously hard. For hardware circuits whose regularity can be succinctly captured by BDDs, $10^{300}$ states or more can be explored in a matter of minutes [27]. However, as pointed out above, software in general and software specifications in particular have been believed not to exhibit such regular structure. In addition, certain specifications are infinite-state and therefore not immediately amenable to conventional model checking. Early successful attempts on software demonstrated the technique’s feasibility only on small examples [10, 137].
1.2 Contributions

Although there are undoubtedly software systems for which BDD-based model checking is inappropriate, the thesis of this work is that the approach can be extremely effective for certain important applications, provided we develop intuitions and techniques to combat the BDD-blowup problem. This dissertation makes three contributions:

Case Studies We carried out two significant case studies of applying BDD-based model checking to portions of the state-machine models of two aircraft systems, namely, the TCAS II system requirements specification mentioned above, and a research-oriented model of an electrical power distribution system used on certain Boeing commercial aircraft [116]. These models are written in RSML and statecharts, respectively. We developed a method to systematically translate a class of RSML and statecharts models into inputs to SMV [111], a popular BDD-based model checker.

Though written in languages with similar flavors, the two models differ greatly in the details. A lot of TCAS II’s complexity stems from its nontrivial arithmetic guarding conditions over numeric inputs like the aircraft’s altitudes and altitude rates. The control flow, however, was deliberately designed to be extremely simple [108]. The EPD model, on the other hand, has only Boolean inputs, but its synchronization structure is less trivial. However, in both case studies, we discovered subtle but important flaws in the model that were not disclosed by other techniques such as inspection, consistency checking, or simulation. We also found that model checking is not only useful for verification or falsification, but also helps the user understand the model, such as discovering implicit assumptions made in a component. We have not yet analyzed the entire specifications, but this just shows that it is not necessary to check a complete specification to get significant benefits from the technique.

Optimizations Despite the encouraging final results, the initial experience was daunting: A lot of the properties could not be analyzed within hours of CPU time and hundreds of megabytes of memory because of the enormous BDDs produced. We developed intuitions about some reasons for the BDD-blowups in our domain, and devised various optimization techniques to dramatically reduce the BDD size, thereby making the analyses feasible.

Some of the techniques are quite different from those commonly used in hardware verification, because our techniques are designed to take advantage of the syntax and semantics of the languages: In RSML and statecharts, state transitions are triggered by events and guarded by conditions on control states and environmental inputs. The state machines respond to an environmental event by taking a number of transitions called microsteps. The synchrony hypothesis, borrowed from synchronous programming languages [13], assumes that this whole collection of microsteps, called a macrostep, is atomic with respect to the environment. This distinction between microsteps and macrosteps, together with the synchrony hypothesis, allows the specifier to break down a complex response into a sequence of simpler reactions. Most of the techniques developed in this dissertation attempt to exploit the event-based synchronization, the macrostep semantics, and the synchrony hypothesis.
Extension While BDDs often handle control structures well, certain data operations,
such as multiplication of integers, provably produce exponential-size BDDs [23]. This is a
major obstacle to analyzing the remaining portion of TCAS II and other reactive embedded
systems that base their state transitions on nonlinear predicates over sensor inputs. We
developed an approach of tightly coupling BDDs and an auxiliary constraint solver to attack
the problem. Separating the arithmetic operations from the BDDs allows us to evade the
otherwise inevitable BDD blowups. Sound and complete for a restricted class of systems,
the technique allows for arbitrarily complicated arithmetic predicates and arbitrary data
domains (such as real-valued inputs), assuming an appropriate solver is available.

These are important steps toward making symbolic model checking a viable verification
technique for state-machine software specifications, particularly those for safety-critical sys-
tems.

1.3 Related Work

This section lists some general related work. Other relevant research will be examined in
subsequent chapters.

There are many other widely-researched approaches to fighting the state-explosion prob-
lem, such as partial-order reduction [64, 134], symmetry reduction [42, 57, 96], composition
[39, 68], abstraction [44], and inequality necessary conditions [48]. Corbett compared three
of these approaches, namely, BDD-based model checking, partial-order reduction, and in-
équality necessary conditions. For detecting deadlock in Ada tasking programs, he observed
that “no technique was clearly superior to the others, but rather each excelled on certain
kinds of programs.” [46, p.179].

Particularly worth discussing is partial-order reduction, because it is often linked with
software verification. The technique exploits the commutativity of independent actions in
asynchronous processes to obtain exponential time and space savings. It has been imple-
menced in the the Spin model checker [85, 86], which is advertised as the model checker
for software. Efficient for verifying many asynchronous software protocols, it deliberately
does not use BDDs but relies on explicit state enumeration. BDD-based algorithms, on the
other hand, have been observed to perform less efficiently for asynchronous (hardware or
software) systems than for synchronous systems [27]. Perhaps because concurrent software
is often associated with asynchrony, many people just take it for granted that BDDs do not
work well for any kind of software, even for synchronous models such as statecharts (for
which partial-order methods do not yield any benefits).

There have been several other recent case studies of model checking for safety-critical
software. Sreemani and Atlee [131] used SMV to analyze the A-7E aircraft software require-
ments. Bharadwaj and Heitmeyer [14] continued this line of work, using SMV and Spin for
other SCR requirements. From an informal specification of a hydroelectric power plant,
Pugliese and Tronci [122] developed a process-algebra specification, which was then verified
with an in-house BDD-based model checker. Crow and Di Vito [51] verified invariants of
the requirements for a software subsystem on the Space Shuttle of NASA with the explicit
model checker Murϕ [54]. Using Spin, Havelund et al. [74] found errors in a space craft
controller, while Schneider et al. [129] validated a model of a fault-tolerant system specified as a hierarchical state machine.

An important difference between this work and theirs is that we used the case studies as vehicles for understanding performance issues and tried to improve the symbolic model-checking algorithms based on the insights learned. The optimizations in Chapters 6 and 7 are results of this process. The modeling languages and the environmental assumptions also differ. For example, their system environments were abstracted as a set of predicates or variables with a small enumerative range, whereas the inputs to our TCAS II model include numerical values. Numerical calculation and comparison are abundant in the TCAS II specification, and they caused significant problems in model checking. The technique in Chapter 8 represents an attack on one of the issues.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter 2 reviews symbolic model checking and the model checker SMV. The syntax and semantics of statecharts and RSML, along with their translation to SMV, are presented in Chapter 3. Chapters 4 and 5 report on our case studies, while Chapters 6 and 7 explain our optimization techniques. The method of combining constraint solving with BDDs is described in Chapter 8. Chapter 9 concludes the dissertation with some future work.
Chapter 2

Symbolic Model Checking

Model checking is an approach to formal verification in which a reactive system modeled as a state transition system is verified against a property specified as a temporal-logic formula. This chapter gives a brief overview of model checking and the symbolic model checker SMV.

2.1 Model Checking

The behaviors of a reactive system can be modeled as a set of states and transitions among the states. Formally, a finite-state transition system is a tuple \((Q, R, I)\), where \(Q\) is a finite set of states, \(I \subseteq Q\) is the set of initial states, \(R \subseteq Q \times Q\) is the transition relation, which is assumed to be total, that is, for each state \(q\), there is always a \(q'\) with \((q, q') \in R\). If we have \((q, q') \in R\), then \(q'\) is a successor of \(q\), and \(q\) is a predecessor of \(q'\). A path is an infinite sequence of states in which each adjacent pair of states belongs to \(R\). A path is a trace if the first state is initial. A state is reachable if it appears on some trace, and a set of states is reachable if some state in the set is.

Given a transition system, we would like to describe desirable system behaviors, or properties, in some concise way. Many properties can be expressed in the propositional Computation Tree Logic (CTL) \([40]\). We refer the reader to Clarke et al. \([41]\) for its precise semantics. Roughly, CTL formulas are built from propositions (predicates over the states), the usual Boolean operators, path quantifiers \(A\) (for all paths) and \(E\) (for some path), and modalities \(X\) (next-time), \(F\) (future, or eventually), \(G\) (global, or henceforth), \(U\) (strong until), and \(W\) (weak until), with every modality immediately preceded by a path quantifier. Each modality is evaluated over a path, and intuitively

- \(X\phi\) means that \(\phi\) holds on the path starting at the next state,
- \(F\phi\) means that \(\phi\) holds somewhere on the path,
- \(G\phi\) means that \(\phi\) holds everywhere on the path,
- \(\phi U \psi\) means that \(\psi\) holds somewhere on the path and \(\phi\) holds everywhere before that, and
- \(\phi W \psi\) means that \(\phi\) holds everywhere before \(\psi\) holds, and \(\phi\) must hold forever if \(\psi\) never holds.
Start with $Y_0 = E$ and iteratively compute $Y_{i+1} = \text{Pre}(Y_i) \cup Y_i$ until reaching a fixed point.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fixed_point.png}
\caption{Algorithm for computing $\text{Pre}^*(E)$}
\end{figure}

For example, the formula $\text{AG} \neg \text{error}$ asserts that the error states are not reachable, $\text{AGAF stable}$ asserts that the system is stable infinitely often, $\text{AG} (\text{request} \rightarrow \text{AFack})$ asserts that a request is always followed by an acknowledgement in the future, $\text{AG} (\text{request} \rightarrow \text{A(request Wack)})$ asserts that once issued, a request will persist unless an acknowledgement is given, and $\text{AGEF restart}$ asserts that it is always possible to restart.

The model-checking problem is to determine whether a given CTL formula is satisfied by a given transition system. If the formula does not hold, a model checker typically tries to find a counterexample, a trace that falsifies the formula.

To illustrate how model checking is performed, consider the CTL formula $\text{AG} \neg \text{error}$, the simplest and the most common kind of temporal property checked in practice. We define $\text{Pre} : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ to be the predecessors of a set of states under the transition relation $R$:

$$\text{Pre}(S) = \{ q \in Q \mid \exists q' \in S. \ (q, q') \in R \}. $$

Intuitively, it is the set of states that may reach some state in $S$ in one transition. Then we can characterize the model-checking problem for $\text{AG} \neg \text{error}$ in a set-theoretic manner: Determine whether $I \cap \text{Pre}^*(E)$ is empty, where $E$ is the set of error states (states in which the proposition error holds) and $\text{Pre}^*(E)$ is the set of states that may eventually reach an error state. More specifically, it is the least fixed point of $E \cup \text{Pre}(Y)$, or the smallest state set $Y$ that satisfies of the equation $Y = E \cup \text{Pre}(Y)$. The existence of the fixed point is guaranteed by the finiteness of $Q$ and the monotonicity of $\text{Pre}$. Figure 2.1 shows an iterative algorithm for computing this fixed point. The set $Y_i$ is the set of states that may reach an error state in at most $i$ transitions. All other CTL formulas can be computed similarly as (possibly multiple) fixed points.

### 2.2 Binary Decision Diagrams

In so-called “explicit” model checking, a state set is represented simply by labeling individual states in the transition system [41, 123]. The method is impractical for many large systems
because of the state-explosion problem. Much more efficient for large state spaces is symbolic model checking, in which the model checker visits the whole set of states at the same time [28, 49, 111].

The main idea is to represent a state set symbolically as a Boolean function. Notice that the state space $Q$ can be represented by a finite set $X$ of variables, such that each state in $Q$ corresponds to a valuation for the variables and no two states correspond to the same valuation. For finite-state systems, we can assume without loss of generality that each variable is Boolean. A set $S$ of states is then symbolically represented as a Boolean function $S(X)$ such that a state is in the set if and only if it makes the function true. The transition relation of states can be similarly represented as a Boolean function $R(X, X')$ where $X'$ is a copy of $X$ and represents the next state. Intersection, union, and complementation on sets or relations respectively becomes conjunction, disjunction, and negation on Boolean functions. Now the problem of representation of state sets is reduced to that of Boolean functions.

Empirically, the most efficient representation for Boolean functions is reduced ordered binary decision diagrams, or BDDs [22, 24]. Intuitively, a BDD is like a binary decision tree, except that isomorphic subtrees must be combined resulting in a directed acyclic graph. In addition, each path can test a Boolean variable at most once, and must comply with a fixed linear order of the variables. Figure 2.2 shows a BDD for the odd parity function over the Boolean variables $x_0$, $x_1$, and $x_2$. BDDs are canonical, meaning that given a Boolean function and a variable order, the BDD that represents the function is unique. Most operations over BDDs can be performed efficiently. For example, the time and space complexities of computing the conjunction or disjunction of two BDDs are linear in the size of the result, which is at most the product of the sizes of the operands. Negation and equivalence checking can be done in constant time [19].

We can now represent a state set $S$ and the transition relation $R$ as BDDs and compute the predecessors of $S$ as

$$Pre(S) = \exists X'. R(X, X') \land S(X').$$

The notation $\exists X'$ refers to existentially quantifying out all the variables in $X'$. In addition to Boolean operations and equivalence checking, operations like existential quantification

![Figure 2.2: BDD for odd parity. Dotted lines represent 0-branches, while solid lines represent 1-branches.](image-url)
and variable substitution can also be performed, so the model-checking algorithm can be implemented using BDDs. Thanks to the succinctness of BDDs and the efficiency of their algorithms, some systems with over $10^{120}$ states can be analyzed [27].

Although efficient for many synchronous hardware circuits, the BDD size is notoriously sensitive to the variable order and the various parameters chosen in the implementation. Given a description of a transition system and a formula to check, it is extremely hard, theoretically or practically, to predict the size of the BDDs that will be produced.

2.3 SMV

SMV [111] is a CTL symbolic model checker using BDDs to represent state sets and transition relations. Discussions and experimental results in this thesis are based on CMU’s SMV release 2.4.4 (with modifications described in Chapters 6 and 7). Below we summarize the SMV features pertinent to our discussion. In SMV, 1 represents true, and 0, false. The logical operators and, or, and not are &., |, and !, respectively.

An SMV program consists of a description of a finite-state transition system and a list of CTL formulas. Recall that a transition system is defined by a state space, a transition relation, and a set of initial states. The state space is determined by state variable declarations, precede by the keyword VAR. For example, the code

```plaintext
VAR
  b: boolean;
  x: 0..7;
  s: {on, off};
```

declares a Boolean variable b, an integer variable x ranging between 0 and 7, and a variable s with value drawn from the set {on, off}. The variable x is internally represented as three Boolean variables.

The transition relation and the initial states can be specified by a collection of simultaneous assignments: Initial-state assignments are made simultaneously at the start, and subsequently next-state assignments are simultaneously executed once per cycle. Assignments are preceded by the keyword ASSIGN. For any variable var, init(var) refers to the value of var in the initial states, so the code

```plaintext
ASSIGN
  init(b) := 0;
```

sets the initial value of b to 0. To define the transition relation, the expression next(var) represents the value of var in the next states. Therefore

```plaintext
ASSIGN
  next(b) := !b;
```

specifies the next-state value of b to be the negation of its current value; that is, its value toggles between 0 and 1 forever. The next operator can also appear on the right hand side of an assignment.

A common way to define next-state values is to use a case expression:
ASSIGN
next(x) := case
  x<7: x+1;
  1: 0;
esac;

This says that if the value of \( x \) is currently less than 7, it will be incremented by 1 in the next state; otherwise, it will be reset to 0. In other words, \( x \) is a modulo-8 counter. (The branches are evaluated sequentially, and since 1 means true, the second branch represents the default case.)

SMV has a macro-like facility for defining a symbol to represent an expression, using the keyword DEFINE. Notice that a state variable is not introduced for such defined symbols. For example:

DEFINE
d := x=7 & b=0;
ASSIGN
next(s) := case
d: on;
  1: off;
esac;

The code above sets the next-state value of \( s \) to on when \( d \) is true, that is, when \( x \) is 7 and \( b \) is false. The next operator can also be applied to defined symbols (but can only appear on the right hand side of an assignment). That is, \( \text{next}(\text{sym}) \) gives the value of \( \text{sym} \) in the next state. This is equivalent to replacing each variable \( \text{var} \) with \( \text{next}(\text{var}) \) in the definition of \( \text{sym} \). For example, \( \text{next}(d) \) is identical to \( \text{next}(x)=7 & \text{next}(b)=0 \).

Two sources of nondeterminism in SMV are relevant to us. An expression can be a set, and it nondeterministically evaluates to a value from that set. As an example, the code

ASSIGN
init(x) := \{0,1\};

restricts the initial value of \( x \) to either 0 or 1. In addition, when the initial or the next-state value of a variable is not specified, it nondeterministically evaluates to a value of its type.

An alternative way to specify the transition relation is to use the keyword TRANS, followed by an arbitrary expression involving the state variables, defined symbols, and/or their next versions. The expression directly defines the transition relation as a proposition. For example, the assignment to \( \text{next}(s) \) above is equivalent to the following:

TRANS (d & next(s)=on) | (!d & next(s)=off)

Next-state assignments define the transition relation imperatively, whereas TRANS expressions define it declaratively. TRANS expressions are sometimes more succinct and are strictly more expressive. However, they are less robust; for example, an empty transition relation can be specified with TRANS expressions, resulting in strange analysis results. Such problems can be hard to track down, so TRANS expressions must be used with care.

A program can contain both next-state assignments and TRANS expressions. Their conjunction forms the transition relation.
Chapter 3

Statecharts and RSML

Although general, transitions systems as described in Chapter 2 do not provide any high-level constructs, distinguish between control and data, or impose any structure on the system. They are thus too low-level for specifying complex reactive systems. In this chapter, we look at the statecharts language and its variant RSML, which are more appropriate for this purpose. We describe the syntax and intuitive semantics, and see how statecharts and RSML machines can be formally defined as a transition system and as an SMV program. Preliminary results in this chapter appeared in Anderson et al. [9] and Chan et al. [34].

3.1 Syntax and Semantics

The statecharts language is an informally-defined visual state-machine language extending conventional state diagrams with state hierarchies and broadcast communications [71]. RSML is a language based on statecharts with a semantics [107]. Another common variant of statecharts semantics is that of STATEMATE [73]. We will focus on subsets of the two languages, and unless the distinctions are important to our discussion, we will use “statecharts” to refer to either of them.

State Hierarchy  State hierarchies allow the system to have deep and orthogonal structures. More precisely, each state $S$ may contain substrates, whose superstate is $S$. The state $S$ is either an and-state or an or-state. Intuitively, the system is in $S$ if and only if (1) it is an and-state and the system is in all of its substrates, or (2) it is an or-state and the system is in exactly one of its substrates. We use $S \triangleright A$ to denote that a superstate $S$ is in its substate $A$. Each or-state has exactly one default substate; intuitively, if the system enters an or-state, it also enters its default substate unless it explicitly enters some other substate. If a state has no substrates, it is an atomic state. For non-atomic states, we often use the words “machines” and “states” interchangeably.

Figure 3.1 shows an example of a statecharts machine. It only illustrates some of the features of statecharts, and does not represent any real device. It triggers an alarm when the altitude of an aircraft is too low according to certain criteria. The hierarchical structure is shown in the diagram by containment. At the highest level, $Sys$ is an and-state, whose substrates are $Alt-Layer$ and $Alarm$ (substates of an and-state are separated by dashed lines). $Alt-Layer$ is an or-state with three substrates, $High$, $Mid$, and $Low$. $Alarm$ is also an or-state, with substrates Shutdown and Operating; the latter is an and-state, containing Mode and
Figure 3.1: An example of a statecharts machine
Figure 3.2: State hierarchy drawn as tree. The shaded nodes represent and-states, unshaded nodes represent or-states, and the leaves are atomic states.

Volume. Default states, e.g., Mid, are indicated by arrows without origins. Figure 3.2 shows the hierarchy as a tree.

Inputs and Events The example contains two input variables from the environment, namely alt (an integer) and switch (up, down, or test). The input alt represents the altitude of the aircraft, and switch is controlled by the pilot.

States are synchronized by events, which are broadcast to the entire system. There are three events in the example: u, v, and w; the first two are generated by the environment and are called external events. The environment is supposed to generate u periodically, and v is generated when the pilot changes the volume of the alarm. The event w is generated by the statecharts for internal synchronization. For simplicity, we assume in general that an event is either generated externally by the environment or internally by the statecharts, but not both.

Transitions A transition is represented as an arrow originating from a source state to a destination state. We label a transition with the form

\[ id: trig[cond]/acts \]

where id uniquely identifies a transition and is only used for our presentation; trig is a trigger event; the guarding condition cond is a predicate on states and inputs; and acts is a set of action events. The guarding condition and the actions are optional. The idea is that if the machine is in the source state, the trigger occurs, and the guarding condition is true (it is considered true if absent), then the transition is enabled. If no other conflicting transitions are enabled (intuitively two transitions conflict if they cannot be taken at the same time), then this transition is taken: The machine exits the source state, enters the
destination state, and generates the action events. Additional states may be entered or exited to maintain the integrity of the state hierarchy. For example, if $t_{14}$ is taken, the state $On$ is entered, so are $Operating$, $Mode$, $Volume$, and $I$.

The system operates as follows. Initially, the environment generates some external events, enabling transitions as described above. A maximal set of enabled transitions that are mutually non-conflicting is then taken, possibly generating new events. This is called a microstep. (Notice that if there are conflicting transitions enabled, then this maximal set is not unique, resulting in nondeterminism.) After each microstep, all the events except those newly generated vanish. These new events are broadcast to the whole system and may trigger other transitions. This process continues until no more transitions are enabled, at which point the system becomes stable. This cascading of microsteps, from the point when the external events arrive to the point when the system becomes stable, is called a macrostep. RSML assumes the synchrony hypothesis [13], which says that during a macrostep, no new external event may occur and the values of the inputs remain unchanged. In other words, the system runs infinitely faster than the environment. Figure 3.3 illustrates the ideas. Once the system is stable, inputs can change and external events can again occur. The STATEMATE semantics optionally allows the synchrony hypothesis, but does not enforce it. We will assume the synchrony hypothesis throughout this thesis.

**AND/OR Tables** The guarding condition $c$ of transition $t_{30}$, too complex to fit in Figure 3.1, is shown in Figure 3.4 as an AND/OR table, one of the distinguishing features of RSML. The leftmost column of the table shows a list of predicates, and the table represents a proposition over these predicates in disjunctive normal form. That is, each column (except the leftmost one) evaluates to the conjunction of the predicates marked T in that column (or their negations if any were marked F), and the entire table evaluates to true if one or more of its columns is true. In RSML, informally, the expression $\text{Prev}(expr)$ refers to the value of $expr$ at the end of the previous macrostep. Also, the special variable $t$ indicates the current time, while $t(Exital(S))$ is the time when state $S$ was last exited. (Note that the synchrony hypothesis implies that the value of $t$ does not change during a macrostep.) So the table reads: (row 1) the system is in state $Low$, and either (column 1) the current value of $alt$ is less than 1000, or (column 2) both the current and previous values of $alt$ are less than 1500, or (column 3) the system exited $Mid$ at least 5 time units ago.
Transition(s): \( \text{Off} \rightarrow \text{On} \)

Location: Mode

Trigger Event: \( w \)

Condition:

\[
\begin{array}{ccc}
\text{And-Event} & \text{OR} \\
\text{Alt-Layer <= Low} & T & T & T \\
\text{alt < 1000} & T & . & . \\
\text{alt < 1500} & . & T & . \\
\text{Prev} (\text{alt}) < 1500 & . & T & . \\
\text{t} \geq t (\text{Exited} (\text{Mid})) + 5 & . & . & T \\
\end{array}
\]

Output Action:

Figure 3.4: Transition from Off to On

To reduce the sizes of the AND/OR tables, RSML allows functions and macros. For example in Figure 3.4, instead of an input, \( \text{alt} \) could have been a function defined elsewhere, and its value might depend on inputs and states. Similarly, a macro can replace a primitive predicate in the leftmost column, and is defined as an AND/OR table elsewhere. Functions and macros can optionally take parameters.

3.2 Translation Example

In this section, we translate the statecharts example above to SMV. The complete SMV program is shown in Appendix A.1.

**SMV Variables** First, we declare the SMV variables for the state hierarchy, inputs, and events. The events are easy—they are naturally translated to Boolean variables. For example, we have:

```plaintext
VAR
  u: boolean;
```

and similarly for other events. The intended meaning is that the variable is true if and only if the event has just been generated. The input \textit{switch} is also straightforward:

```plaintext
switch: {up, down, test};
```

However, the statecharts do not specify the upper and lower bounds for \( \text{alt} \). Obviously, any lower bound less than 1950 and any upper bound greater than 10050 will be sufficient. In fact, \( \text{alt} \) can be represented by five values, but let’s keep the translation straightforward and define it to range between 0 and 20000 (BDDs should handle this range without problem):

```plaintext
alt: 0..20000;
```
The state hierarchy can be encoded as follows. Each or-state in the hierarchy provides a choice of its substates, so it seems natural to declare a variable for each or-state, whose substates form the range of the variable:

$$\text{Alt-Layer: } \{\text{High, Mid, Low}\};$$
$$\text{Alarm: } \{\text{Shutdown, Operating}\}$$
$$\text{Mode: } \{\text{Off, On}\};$$
$$\text{Volume: } \{1, 2\};$$

The values of these variables completely determine the current states of the system. Note that when the value of \text{Alarm} is \text{Shutdown}, the values of \text{Mode} and \text{Volume} are irrelevant. We find it convenient to define a symbol to indicate the exact condition under which the system is in a particular state. For example, we define

```plaintext
DEFINE
  in-Sys := 1
  in-Alt-Layer := in-Sys;
  in-Alarm := in-Sys;
```

because the system is always in \text{Sys}, and thus always in \text{Alt-Layer} and \text{Alarm} as well. We also have

```plaintext
  in-High := in-Alt-Layer & Alt-Layer = High;
  in-Operating := in-Alarm & Alarm = Operating;
  in-Mode := in-Operating;
  in-Off := in-Mode & Mode = Off;
```

Conditions for other states can be defined similarly.

**Statecharts Transitions** Now we can define when a transition, say \text{t7}, is enabled:

```plaintext
DEFINE
  t7 := in-Mid & u & alt < 1950;
```

This simply reflects the definition: A transition is enabled when the system is in the source state, the trigger event occurs, and the guarding condition is true. Now, to specify the state change, we use the following self-explanatory code:

```plaintext
ASSIGN
  next(Alt-Layer) :=
    case
      t1|t4: High;
      t2|t5|t6: Mid;
      t3|t7: Low;
      1: Alt-Layer;
    esac;
```

We also need to specify the event generated:
next(w) := t1|t2|t3|t4|t5|t6|t7;

and initialize the states and event:

init(Alt-Layer) := Mid;
init(Alarm) := Shutdown;
init(w) := 0;

Note that the values of Mode and Volume in the initial states are irrelevant, so we do not need to initialize them, although initializing them to any value does no harm.

**Inputs**  Unless explicit constraints are given, inputs to the system are modeled nondeterministically to allow arbitrary environmental behaviors. However, the synchrony hypothesis precludes the inputs from changing when the system is not stable. We define what it means to be stable:

\[
\text{DEFINE}
\]

\[
\text{stable} := !(u|v|w);
\]

Then the input \textit{alt} changes according to the following assignment:

\[
\text{ASSIGN}
\]

\[
\text{next(alt)} :=
\begin{cases}
\text{case} \\
\text{stable} & \text{!next(stable): 0..20000;}
\end{cases}
\]

\[
1: \text{alt;}
\]

\[
esac;
\]

The default branch above maintains the synchrony hypothesis by keeping the value of the variable unchanged during a macrostep. The code for \textit{switch} is similar. Events \textit{u} and \textit{v} are also unconstrained at the beginning of a macrostep, but during a macrostep they are never generated:

\[
\text{next(u)} :=
\begin{cases}
\text{case} \\
\text{stable: \{0,1\};}
\end{cases}
\]

\[
1: 0;
\]

\[
esac;
\]

Inputs and external events need not be initialized because they are unconstrained at the start of a macrostep.

**Prev and Timing Constraints**  We have already translated most of the statecharts except two predicates in the AND/OR table in Figure 3.4. Referencing the previous value of \textit{alt} requires the introduction of an extra variable \textbf{prev-alt} to remember its value at the end of a macrostep:
VAR
prev-alt: 0..20000;
ASSIGN
next(prev-alt) :=
case
  stable: alt;
  1: prev-alt;
esac;

Timing constraints are a little tricky. First, we assume that time is discrete. To translate the expression $t \geq t(Exit(Mid)) + 5$, we observe that it is sufficient to know $t - t(Exit(Mid))$, represented by the variable time-Mid below:

VAR
time-Mid: 0..5;
ASSIGN
next(time-Mid) :=
case
t2|t4|t7
   : 0;
stable & time-Mid < 5: time-Mid + 1;
1
   : time-Mid;
esac;

The timer time-Mid indicates the number of time units passed since Mid was last exited. The timer is reset when Mid is exited via transitions t2, t4, or t7. At the end of a macrostep, the value of the timer is incremented unless it is already five—since we only care whether the timer is at least five, specific values greater than five are irrelevant.

Note that taking t2 is viewed as exiting and re-entering Mid. However, if t2 were specified as a so-called identity transition in RSML, then taking the transition would not reset the timer. In that case, we would simply leave out t2 in the first case branch. For simplicity, we will not further discuss identity transitions in this chapter.

3.3 Translation Rules

To explain the translation from statecharts to SMV more generally and precisely, we first formally define the statecharts as a state transition system introduced in Section 2.1, based on the operational semantics of RSML and STATEMATE. Some of our definitions are based on Pnueli and Shalev [121]. For simplicity, we first assume the absence of timing constraints and Prev functions. Then we show how we translate deterministic statecharts machines and certain nondeterministic machines to SMV programs. Timing constraints and Prev functions are considered later in this section, along with some discussions on alternative semantics and translation rules.

3.3.1 Statecharts as Transition Systems

We define statecharts as a transition system $(Q, R, I)$. To distinguish between a statecharts state and an element in $Q$, we call the former a local state and the latter a global state when there are ambiguities. The relation $R$ is thus called the global transition relation.
Local States  Let \( \text{States} \) be the finite set of local states, and let \( \text{Children} : \text{States} \rightarrow \mathcal{P}(\text{States}) \) map each local state to its substates, or children. The function is required to impose a tree structure on the local states with a distinguished element \( \text{root} \) as the root of the tree. We sometimes use \( \text{parent} \) as synonym for superstate.

We define \( \text{Children}^+ \) and \( \text{Children}^* \), the transitive and reflexive-transitive closures of \( \text{Children} \), as

\[
\begin{align*}
\text{Children}^+ &= \bigcup_{i \geq 1} \text{Children}^i \\
\text{Children}^* &= \bigcup_{i \geq 0} \text{Children}^i,
\end{align*}
\]

where for each state \( p \) in \( \text{States} \),

\[
\begin{align*}
\text{Children}^0(p) &= \{p\} \\
\text{Children}^{i+1}(p) &= \bigcup_{s \in \text{Children}(p)} \text{Children}^i(s) \quad \text{for } i \geq 0.
\end{align*}
\]

If \( s \in \text{Children}^*(p) \), then we say that \( s \) is a descendant of \( p \), that \( p \) is an ancestor of \( s \), and that \( s \) and \( p \) are ancestrally related. If in addition \( s \neq p \) (that is, \( s \in \text{Children}^+(p) \)), then \( s \) is a strict descendant of \( p \), and \( p \) is a strict ancestor of \( s \). If \( \text{Children}(s) = \emptyset \), then \( s \) is an atomic state. Otherwise, it is either an and-state or an or-state; in the latter case, it has exactly one default substate.

Intuitively, a configuration is a maximal set of local states that the system can be in simultaneously. That is, \( C \subseteq \text{States} \) is defined to be a configuration if

1. \( \text{root} \in C \);
2. for every and-state \( s \), either \( s \) and all substates of \( s \) are in \( C \), or they are all not in \( C \); and
3. for every or-state \( s \), either \( s \) and exactly one substate of \( s \) are in \( C \), or \( s \) and all substates of \( s \) are not in \( C \).

For example, in Figure 3.1, the set of states \( \{\text{Sys, Alt-Layer, High, Alarm, Shutdown}\} \) is a configuration.

Global States  Let \( \text{Config} \subseteq \mathcal{P}(\text{States}) \) be the set of all configurations, \( \text{Events} \) be the finite set of events, and \( \text{Inputs} \) be the set of all possible assignments to the input variables. The set \( Q \) of global states is defined to be \( \text{Config} \times \mathcal{P}(\text{Events}) \times \text{Inputs} \). In other words, a global state is a triple consisting of a configuration, a set of events, and an assignment to the input variables.

Initial Global States  Intuitively, the default completion of a local state \( p \), denoted \( \text{Complete}(p) \), is the unique configuration \( C \) containing \( p \) such that for any or-state, its default substate is preferred over other substates. That is, for each or-state \( s \in C \) that is not a strict ancestor of \( p \), the default substate of \( s \) is also in \( C \). For example, the default completion of \( \text{On} \) is \( \{\text{Sys, Alt-Layer, Mid, Alarm, Operating, Mode, On, Volume, 1}\} \).

Let \( \text{External} \subseteq \text{Events} \) be the set of external events. The set \( I \) of initial global states is the set of every triple \((C, E, V)\) with \( C = \text{Complete}(\text{root}) \), \( E \subseteq \text{External} \), and \( V \in \text{Inputs} \).
Local Transitions  Let Trans be the set of local transitions. Each transition $t \in Trans$ has five attributes: the source state $src(t) \in States$, the destination state $dest(t) \in States$, the trigger event $trig(t) \in Events$, the guarding condition $cond(t) \subseteq \mathcal{P}(States) \times Inputs$, and the action events $acts(t) \subseteq Events \setminus External$.

The scope of a local transition $t$, denoted by $scope(t)$, is defined as the lowest common strict or-ancestor of the source and the destination; that is, $scope(t)$ is an or-state that is a strict ancestor of both $src(t)$ and $dest(t)$, and every such or-state is an ancestor of $scope(t)$. The scope of a transition is visualized in the state diagram as the smallest or-state strictly containing both the source and the destination, and intuitively is the minimal context of the transition. We require that each local transition in Trans must have a well-defined scope, making, for instance, any transition out of root illegal. For example, the scopes of $t_1$ through $t_7$ are Alt-Layer, and the scope of $t_{14}$ is Alarm.

Global Transitions  A local transition $t$ is enabled in a global state $(C, E, V)$ with $C \in Config$, $E \subseteq Events$ and $V \in Inputs$, if $src(t) \in C$ (the system is in the source) $trig(t) \in E$ (the trigger occurs), and $(C, V) \in cond(t)$ (the guarding condition holds).

We say that two distinct local transitions conflict if their scopes are ancestrally related. For example, the transitions in Alt-Layer (that is, $t_1$ through $t_7$) are pairwise conflicting since their scopes are identical and thus ancestrally related. Transitions $t_9$ and $t_{10}$ also conflict, because the scope of $t_9$ (Alarm) is an ancestor of the scope of $t_{10}$ (Mode).

Define $maxsrc(t)$ to be the unique child of $scope(t)$ that is an ancestor of $src(t)$, and $maxdest(t)$ to be the unique child of $scope(t)$ that is an ancestor of $dest(t)$, For instance, $maxsrc(t_{14})$ and $maxdest(t_{14})$ are Shutdown and Operating respectively. If a transition $t$ is taken, all descendants of $maxsrc(t)$ that the system is currently in are exited, and certain states, descendants of $maxdest(t)$ induced by $dest(t)$, are entered.

Formally, for any transition $t$, we define $Exits(t)$ as $Children^*(maxsrc(t))$ and $Enters(t)$ as the intersection of $Complete(dest(t))$ and $Children^*(maxdest(t))$. The set $Enters(t)$ precisely specifies the states that the system enters on taking transition $t$. The set $Exits(t)$ is a little less precise. Clearly, before transition $t$ is taken, the system is in some states in $Exits(t)$, in particular $src(t)$, and after $t$ is taken the system is no longer in any state in $Exits(t)$. The mere fact that $t$ is taken does not in general specify any more information than this about the states that the system is in prior to the transition. As an example, $Exits(t_{11})$ is $\{Shutdown\}$, $Enters(t_{11})$ is $\{Operating, Mode, On, Volume, 1\}$, $Exits(t_9)$ is all the descendants of Operating, and $Enters(t_9)$ is $\{Shutdown\}$.

The global transition relation

$$R \subseteq (Config \times \mathcal{P}(Events) \times Inputs)^2$$

is defined as the set of tuples $(C, E, V, C', E', V')$ such that there exists a set of local transitions $T \subseteq Trans$ satisfying all of the following:

(a) Every transition in $T$ is enabled in $(C, E, V)$.

(b) No two transitions in $T$ conflict.

(c) $T$ is maximal: Every transition not in $T$ but enabled in $(C, E, V)$ conflicts with some transition in $T$. 
(d) \( C' = (C - \bigcup_{t \in T} \text{Exits}(t)) \cup \bigcup_{t \in T} \text{Enters}(t) \).

(e) If \( T \neq \emptyset \), then \( E' = \bigcup_{t \in T} \text{acts}(t) \) and \( V = V' \).

(f) If \( T = \emptyset \), then \( E' \subseteq \text{External} \) and \( V' \in \text{Inputs} \).

The transitions in \( T \) are said to be taken. If in some reachable global state the choice of \( T \) is not unique, then the system is nondeterministic. Point (e) above generates the action events and keeps the input variables unchanged according to the synchrony hypothesis, while Point (f) generates a subset of external events and assigns new values to the inputs, indicating the end of a macrostep.

### 3.3.2 Translating Global States

Recall that a global state consists of a configuration, a set of events, and an assignment to inputs. We assume that the numbers of local states, events, and inputs, as well as the range of each input are all finite, so the global state space is also finite. To symbolically encode the events, we declare a Boolean variable for each of them. Similarly, a naive encoding of the configurations, each being a set of local states, is to declare a Boolean variable for each local state (this encoding is used by Philipps and Yoneda [119]). The encoding can be improved by the observation that a configuration is uniquely determined by its intersection with the set of atomic states. So we only need a Boolean variable for each atomic state. This method, however, still requires a large number of Boolean variables—an or-state with \( n \) atomic substates requires \( n \) Boolean variables.

The optimal encoding for this or-state is obviously to declare one variable with a range of size \( n \) (or equivalently declare \( \log n \) Boolean variables). The encoding described in Section 3.2 is a natural extension of this idea. Recall that for each or-state \( s \), we declare a variable with range \( \text{Children}(s) \). To obtain a more succinct encoding, we also flatten nested or-states, i.e., or-states whose superstates or substates are also or-states. For example, taken from the TCAS II requirements (discussed in more detail in Chapter 4), Figure 3.5 shows an example of nested or-states—Composite-RA, RA, and Positive are all or-states. (The vertical bar on the right together with the arrows attached to it is a transition bus, implying a transition between every pair of states connected to the bus. That is, No-RA, Climb, Descend, and Negative are pairwise connected by a transition in either direction.)

Our translated SMV program contains the following code:

```plaintext
VAR Composite-RA: {No-RA, Climb, Descend, Negative};
Climb-VSL: {No-Climb-VSL, VSL0, ... };
Descend-VSL: {No-Descend-VSL, VSL0, ... };
DEFINE in-RA := in-Positive | in-Negative;
    in-Positive := in-Climb | in-Descend;
    in-Climb := in-Composite-RA & Composite-RA = Climb;
```

More generally, let \( O \) be the set of or-states whose parents (if any) are and-states, and let \( A \) be the set of and-states or atomic states that have or-state parents. That is, the set \( O \cup A \subseteq \text{States} \) consists of states at the upper boundaries of the alternations between or-states and and-states in the state hierarchy. Note that root is not contained in \( A \), but
may or may not be in $O$. For each $s \in A$, its leader, denoted as $leader(s)$, is its lowest ancestor in $O$; and for each $p \in O$, the set of its followers, denoted as $Followers(p)$, consists of every $s \in A$ whose leader is $p$. Note that this leader-followers relationship is identical to the parent-child relationship, when there are no nested or-states and nested and-states, and every atomic state has an or-state as parent, as is the case in our example in Figure 3.1. In Figure 3.5, the followers of Composite-RA are No-RA, Climb, Descend, and Negative, while those of Climb-VSL are its children.

Figure 3.6 shows the general SMV code for declaring and initializing the global state variables. As shown in Rule 1, we declare a variable for each leader in $O$, and the range is its followers. For each or-state $s$, let $default(s)$ be its default child, and let $default^*(s)$ be recursively defined as $default^*(default(s))$ if $default(s)$ is an or-state, or $default(s)$ otherwise. (Alternatively, we can characterize $default^*(s)$ as the unique state in $Complete(s) \cap Followers(s)$.) Rules 2–6 tell when the system is in a particular local state. Note that each local state corresponds to exactly one of these five rules, and there are no loops in the recursive definitions. This encoding scheme is one-to-one, and every valuation for the variables corresponds to some legal configuration.

Rules 7–9 are for events and inputs. The set $Range(y)$ denotes the range of the input $y$.

### 3.3.3 Translating Deterministic Transitions

Deterministic statecharts machines are easy to translate because the set of local transitions taken is exactly the set of local transitions that are enabled. Figure 3.7 shows translation rules that are correct only for deterministic machines. If these rules are applied to nonde-
1. For each $p \in \mathcal{O}$:
   \begin{enumerate}
   \item VAR $p$: Followers($p$);
   \item ASSIGN init($p$) := default"($p$);
   \end{enumerate}
2. DEFINE in-root := 1;
3. For each $s \in \mathcal{A}$:
   \begin{enumerate}
   \item DEFINE in-s := in-leader($s$) & leader($s$) = s;
   \end{enumerate}
4. For each and-state or atomic state $s \notin \mathcal{A}$ with parent $p$:
   \begin{enumerate}
   \item DEFINE in-s := in-p;
   \end{enumerate}
5. For each $s \in \mathcal{O}$ with parent $p$:
   \begin{enumerate}
   \item DEFINE in-s := in-p;
   \end{enumerate}
6. For each or-state $p \notin \mathcal{O}$:
   \begin{enumerate}
   \item DEFINE in-p := \bigwedge_{s \in \text{children}(p)} \text{in-s};
   \end{enumerate}
7. For each $e \in \text{Events}$:
   \begin{enumerate}
   \item VAR $e$: boolean;
   \end{enumerate}
8. For each $e \in \text{Events} \setminus \text{External}$:
   \begin{enumerate}
   \item ASSIGN init($e$) := 0;
   \end{enumerate}
9. For each input variable $y$:
   \begin{enumerate}
   \item VAR $y$: Range($y$);
   \end{enumerate}
10. For each $t \in \text{Trans}$:
    \begin{enumerate}
    \item DEFINE $t$ := in-source($t$) & trig($t$) & cond($t$);
    \end{enumerate}
11. For each $t \in \text{Trans}$:
    \begin{enumerate}
    \item DEFINE $t$-taken := $t$;
    \end{enumerate}
12. For each $p \in \mathcal{O}$:
    \begin{enumerate}
    \item ASSIGN next($p$) := case
      \begin{align*}
      t$-taken$ := s; & \quad \text{for each } t \in \text{Trans} \text{ with } & \\
      \text{Followers}(p) \cap \text{Enters}(t) = \{s\} & \text{1: } p; \\
      \text{esac};
      \end{align*}
    \end{enumerate}
13. For each $e \in \text{Events} \setminus \text{External}$:
    \begin{enumerate}
    \item ASSIGN next($e$) := $\bigvee_{t \in \text{trig}(t)} t$-taken;
    \end{enumerate}
14. For each $e \in \text{External}$:
    \begin{enumerate}
    \item ASSIGN next($e$) := case
      \begin{align*}
      \text{stable}: & \{0,1\}; \\
      1: & 0; \\
      \text{esac};
      \end{align*}
    \end{enumerate}
15. For each input variable $y$:
    \begin{enumerate}
    \item ASSIGN next($y$) := case
      \begin{align*}
      \text{stable} & \& \neg \text{next(stable)}: \text{Range}(y); \\
      1: & y; \\
      \text{esac};
      \end{align*}
    \end{enumerate}
16. DEFINE stable := $\neg \bigvee_{e \in \text{Events}} e$;

Figure 3.6: Rules for declaring and initializing SMV variables for statecharts

Figure 3.7: Rules for translating deterministic statecharts transitions
terministic machines, the behavior of the translated SMV program will be identical to the statecharts machines up to the point when some conflicting transitions are simultaneously enabled; after that, the two systems will start to exhibit diverging behaviors.

Rule 10 defines when a local transition is enabled, and Rule 11 defines when it is taken. Because they are the same for deterministic systems, in Section 3.2 and Appendix A.1, we use $t$ instead of $t$-taken for simplicity. The expression $\text{cond}(t)$ in the figure refers to the proposition that describes the guarding condition.

Rule 12 defines the effects of a transition on the state hierarchy. The set $\text{Followers}(p) \cap \text{Enters}(t)$ contains the follower of $p$ that the system enters upon taking $t$. Note that this set is either empty or a singleton set.

Rules 13–16 generate the appropriate events and update the inputs, during and at the end of a macrostep.

We argue informally for the correctness of the translation. Clearly, the rules ensure that only enabled transitions can cause state change or generate action events. We claim that every enabled transition causes the necessary state change: In Rule 12, because of the deterministic assumption, at most one non-default case branch can be true, so each enabled transition $t$ always results in updating every variable that has a follower in $\text{Enters}(t)$. By virtue of our state encoding, this implies that every state in $\text{Enters}(t)$ is entered and every state in $\text{Exits}(t)$ previously occupied is exited. Finally, it is easy to see that enabled transitions always result in the generation of their action events, and that inputs are updated correctly.

Note that the symbol stable, which indicates the end of a macrostep, is defined as $\neg \bigvee \epsilon \in \text{Events} \epsilon$, but a direct translation from the definition of $R$ would be $\neg \bigvee t \in \text{Trans} t$. The two versions are nearly identical, because if no events are occurring, there will be no enabled transitions, and if there are no enabled transitions, no events will occur in the next microstep. Defining stable using events is more concise, as there are usually far fewer events than transitions.

The example in Section 3.2 was translated mostly based on these rules. Careful readers may notice that the system is actually nondeterministic, so the translation is not exact. In Section 4.3.1, we will discuss how to discover the violating transitions.

### 3.3.4 Translating Nondeterministic Transitions

In principle, translating machines with arbitrary nondeterministic transitions is straightforward. One strategy is to declare a set of auxiliary Boolean variables representing the transitions in $\text{Trans}$. We can translate the definition of the global transition relation $R$ literally as a first-order logic formula over the finitely many auxiliary and global state variables, and then optionally quantify out the auxiliary variables. This conceptually simple method is inefficient, because the number of transitions, and thus the number of auxiliary variables, is usually large. There are other potentially more efficient ways of constructing the global transition relation, but in general the constructions are still expensive. Interested readers are referred to the work of Helbig and Kelb [80] for an example.

We now give modifications to the rules in Figure 3.7 to handle a rich class of nondeterministic machines, namely those with the property that a transition is taken if and only if it
11'. For each \( t \in \text{Trans} \):

\[
\text{DEFINE} \quad t\text{-taken} := t \land \bigwedge_{s \in \text{Rules}(t)} \text{next}(\text{in}-s) \land \bigwedge_{e \in \text{Exts}(t)} \text{next}(e);
\]

12'. For each \( p \in \mathcal{O} \):

\[
\text{TRANS} \quad (p = \text{next}(p)) \lor \bigvee_{\text{Path} = \{p\} \cap \text{Exts}(t)} \neg t\text{-taken}
\]

17'. For each \( t \in \text{Trans} \):

\[
\text{TRANS} \quad \neg t \lor \left( \neg t\text{-taken} \iff \bigvee_{t' \in \text{Conflict}(t)} t'\text{-taken} \right)
\]

Figure 3.8: Rules for translating a class of nondeterministic statecharts transitions

\( t \)-taken appears to be taken. Rule 11' in Figure 3.8, replacing Rule 11, explains more precisely what this assumption means: A transition is taken if and only if it is enabled and in the next microstep the system enters the appropriate states and generates the appropriate action events. We will see shortly why this may not be true in general.

Rule 12' and Rule 17' replace Rule 12. Rule 12' ensures that the system remains in a state unless some transition exiting that state is taken. The set \( \text{Conflict}(t) \) is defined as the transitions that conflict with \( t \). Rule 17' concisely says that the set of transitions taken must be maximal and nonconflicting: Either a transition is not enabled (in which case it cannot be taken), or it is not taken because one of the conflicting transitions is, or it is taken, in which case none of the conflicting transitions is. These rules are correct, if the definition of \( t\text{-taken} \) is correct to begin with.

Figure 3.9 shows why the latter is not always true. If events \( x \) and \( y \) can occur simultaneously when the machine is in state \( A \), then the two conflicting transitions are enabled. According to the semantics, the machine will take exactly one of \( t_x \) and \( t_y \), go to state \( B \), and generate event \( z \). However, if we just look at the state change and the event generated, the machine will appear to have taken both \( t_x \) and \( t_y \), making our definition of \( t\text{-taken} \) incorrect. In fact, in this case, our translation prevents the machine from entering state \( B \), because otherwise, Rule 11' would make both \( t_x\text{-taken} \) and \( t_y\text{-taken} \) true, which is precluded by Rule 17'. (On the other hand, Rule 17' also prevents the machine from staying at state \( A \), resulting in deadlock.)

A necessary and sufficient condition for the correctness of this translation is that, for any conflicting transitions that may be simultaneously enabled, the current and next configurations...
tions, together with the set of actions of the system, gives enough information to determine which of these transitions is taken.

Simpler Nondeterminism Although the class of machines captured above is quite rich and the translation does not introduce auxiliary variables, the resulting SMV program can be large because Rule 17 produces code with size quadratic in the number of transitions in the worst case. Furthermore, defining the transitions with the TRANS construct in SMV is more error-prone than using ASSIGN.

However, certain nondeterminism is easy to model. As an example, suppose we want to specify the state Alt-Layer in Figure 3.1 on page 14 as an entirely nondeterministic machine: All the guarding conditions on $t_1$ through $t_7$ are omitted. The translation to SMV in this case is trivial:

\[
\text{ASSIGN} \\
\text{next(Alt-Layer) :=} \\
\text{case} \\
\quad u: \{\text{High, Mid, Low}\}; \\
\quad 1 : \text{Alt-Layer}; \\
\text{esac;} \\
\text{next(w) := u;}
\]

As another example, if the guarding conditions of $t_1$ and $t_5$ were not mutually exclusive, we could use the following code to allow for nondeterminism:

\[
\text{ASSIGN} \\
\text{next(Alt-Layer) :=} \\
\text{case} \\
\quad t1\&t5 : \{\text{High, Mid}\}; \\
\quad t1\&t4 : \text{High}; \\
\quad t2\|t5\|t6 : \text{Mid}; \\
\quad t3\|t7 : \text{Low}; \\
\quad 1 : \text{Alt-Layer}; \\
\text{esac;}
\]

We used assignments similar to these in our case studies and they proved to be sufficient for our experiments.

3.3.5 Translating Timing Constraints

RSML allows the guarding conditions to reference the current time ($t$) or the time any state $s$ was last entered ($\text{Entered}(s)$) or exited ($\text{Exited}(s)$). However, since time grows without bound, the underlying state transition system in general has an infinite number of global states and BDD-based model checking becomes inapplicable.

Fortunately, many common cases can be handled. If we restrict the predicates involving time to comparing $t - \text{Entered}(s)$ or $t - \text{Exited}(s)$ with a constant, then all we need to do is to keep track of such time lapses with variables, which we call timers. Because there can only be finitely many such time predicates, for each timer there exists a largest constant against
which it is compared. So the range of the timer can be bounded by a constant, which is how we translated the example in Section 3.2. More generally, take a timer \( \theta = t - \text{Entered}(s) \) for example. Let \( k_\theta \) be the upper bound, and \( T_\theta \) be the set of transitions \( t \) with \( s \in \text{Enters}(t) \). We have the following code:

\[
\text{VAR} \\
\theta : 0..k_\theta; \\
\text{ASSIGN} \\
\text{next}(\theta) := \\
\text{case} \\
\text{\( \bigvee_{t \in T_\theta} t : 0; \)} \\
\text{stable \&} \; \theta < k_\theta : \theta+1; \\
1 : \theta; \\
esac;
\]

Notice that we do not initialize the timer. When a state \( s \) has not been entered, for example, the value of \( \text{Entered}(s) \) is undefined. We can catch references to undefined values by including in the range of the timer a special symbol that indicates that the timer is undefined, and by initializing the timer to this symbol. A reachability analysis can then tell whether the system may reference the timer before it is defined. Here, since catching such references is not our major concern, we simply leave the initial value unconstrained, and let the model checker search for an initial value that leads to violation of the property being checked.

Comparing two times, like \( \text{Entered}(s_1) > \text{Exited}(s_2) \), can also be handled by introducing extra variables. Although we have been assuming the discrete-time model (i.e., time is a natural number), it is possible to extend model checking to handle the dense-time model (i.e., time is a nonnegative real number), when we restrict to the same class of time predicates \cite{3}. However, when \( t, \text{Entered}(s), \text{Exited}(s) \) are used in arbitrary arithmetic expressions, whether discrete time or dense time is used, the system cannot be precisely modeled as a finite-state system, and in fact, the model checking problem becomes undecidable \cite{4}.

3.3.6 Translating Previous Values

When the value of \( \text{Prev}(y) \) for some input \( y \) is needed, we use the following code:

\[
\text{VAR} \\
\text{prev-y: Range}(y); \\
\text{ASSIGN} \\
\text{next(prev-y) :=} \\
\text{case} \\
\text{stable: y;} \\
1 : \text{prev-y}; \\
esac;
\]

Again, we do not initialize \( \text{prev-y} \) for the same reason that we do not initialize timers. The translation can be easily modified if \( y \) is a state, a macro, or a function.
An alternative translation for \( \text{Prev}(y) \) is to remember the truth values of the predicates involving \( \text{Prev}(y) \) instead of the value of \( \text{Prev}(y) \) itself. For example, for the predicate \( \text{Prev}(alt) < 1500 \) in Figure 3.4, we could remember the truth value of \( alt < 1500 \) instead of the numeric value of \( alt \). This method has the advantage of possibly using fewer BDD variables, but is less general. For example, it cannot deal with predicates involving both previous and current values, like \( \text{Prev}(alt) < alt \).

3.3.7 Miscellaneous

Other RSML Constructs We have not exhausted all RSML constructs, but the rest are easy to translate: Macros and functions without arguments can be translated simply as defined symbols. Those with arguments can be translated as SMV modules, which are analogous to templates and can be instantiated at each call site of the macros or functions. RSML state-machine arrays give a succinct representation for isomorphic substrates of an and-state. They can be translated to SMV as an array of module instances. We also have not detailed the translation of conditional connectives (\( @ \) in Figure 3.5), which, roughly speaking, factor out common triggers or guarding conditions of a set of transitions. The conceptually simplest translation is to remove a conditional connective by conjoining each pair of incoming and outgoing transitions.

Condition-Driven Transitions STATEMATE and some other variants of statecharts allow local transitions not guarded by events. That is, a transition can have labels of the form \( [\text{cond}] / \text{acts} \), where \( \text{cond} \) is a guarding condition and \( \text{acts} \) is a list of action events. Such transitions are enabled when the system is in the source local state and \( \text{cond} \) is true. We call these local transitions condition-driven. Intuitively, instead of checking the guarding condition only when triggered by an event, condition-driven transitions continuously poll the guarding condition.

Handling condition-driven transitions require only simple modifications to our translation rules: We just ignore the conjunct \( \text{trig}(t) \) in Rule 10 on page 25 if \( t \) is condition-driven. We also need to strengthen \( \text{stable} \) in Rule 16 as

\[
(\neg \bigvee_{e \in \text{Events}} e) \land (\neg \bigvee_{e \in \text{CondTrans}} t)
\]

where \( \text{CondTrans} \) is the set of condition-driven transitions. (Section 6.3.4 will give a more efficient encoding for condition-driven transitions.)

STATEMATE Semantics and Constructs STATEMATE does not insist on the synchrony hypothesis but provides it as an option. We can easily forsake the synchrony hypothesis by changing Rule 16 in Figure 3.7 to set \( \text{stable} \) to 1. STATEMATE also provides internal variables and allows assignments to them as actions. In addition, certain transitions that are considered conflicting here are assigned different priorities and thus do not result in nondeterminism when simultaneously enabled. Slight modifications to the rules would suffice for these differences. Other constructs like history connectors and synchronization through activities would require new translation rules.
Granularity of Global Transitions Note that when we defined the global transition relation in Section 3.3.1, we implicitly assumed that a global transition represents a microstep, which seems a natural choice.

Alternatively, a global transition can represent a macrostep. This may be more natural if we are only interested in the stable states, although analyzing properties within a macrostep becomes impossible. In addition, we need to perform a number of analyses at translation time, such as ensuring each macrostep does eventually terminate. The efficiency of model checking is also affected: This representation may blow up the BDD size, but reduces the number of search iterations needed in the model-checking algorithms. Therefore, it is not clear a priori whether this method works better or worse. In our initial TCAS II experiments to be reported in Chapter 4, it resulted in huge BDDs and poor performance, and we have not considered this method further in our case studies.

Yet another possibility is to represent a microstep as a series of global transitions, which directly corresponds to the semantics of RSML given by Leveson et al. [107]. There, instead of being defined as a maximal set \( T \) of nonconflicting transitions, a microstep is equivalently defined by a loop: \( T \) is initially empty, and enabled transitions are added to \( T \) one at a time until a maximal set is obtained. One may therefore choose to represent each iteration in this loop as a global transition. An obvious drawback is the increased number of global transitions required to encode a microstep. However, a more serious problem is the introduction of asynchrony into the model: Even if \( T \) is unique (that is, the microstep is deterministic), there are in general many different orders of picking the transitions in \( T \), and the model checking algorithm will need to explore all these possibilities. Our representation of microsteps can be viewed as a way of statically eliminating such asynchrony.

Alternative Semantics The semantics of statecharts defined by Pnueli and Shalev [121] are quite different from the semantics considered here. It is unclear how to translate from their semantics in a simple way without introducing many auxiliary global state variables.

The RSML semantics defined by Heimdahl and Leveson [77] are slightly different from the semantics considered here, which are based the earlier work of Leveson et al. [107]. The differences become important when conflicting transitions with different triggers are simultaneously enabled, which does not happen in the portion of the TCAS II requirements machines that we modeled. In general, however, different translation rules would be required.

3.4 Discussion and Related Work

Helbig and Kelb [80] gave a BDD encoding for statecharts. The version of statecharts semantics and the state hierarchy encoding that they used are similar to ours, except that they did not assume the synchrony hypothesis and did not flatten nested-or states (Section 3.3.2). With a custom-built BDD-based model checker, they encoded the transition relation more generally than we did to allow for arbitrary nondeterministic transitions, at the expense of the construction cost of the BDDs. We, in contrast, focused on deterministic transitions and certain nondeterministic ones that are easy to model and sufficient for our experiments.
Note that other than preserving it in the translation to SMV, we did not attempt to exploit the state hierarchy in the verification. Indeed, given the semantics, it is unclear how we can further take advantage of the hierarchy to improve the performance: Contrary to what a novice statecharts user might believe, the state hierarchy in statecharts does not hide details or encapsulate information. In particular, because events are always broadcast to the whole system, in general we cannot reason about a substate locally. Alur and Yannakakis [7] give algorithms for exploring hierarchical states, but not in the context of statecharts. Their method does not deal with concurrency, and relies on the multiple instantiations of a module in the hierarchy to obtain time savings.
Case Study: TCAS II

The Traffic Alert and Collision Avoidance System II (TCAS II) is an airborne collision avoidance system required by the United States Federal Aviation Administration (FAA) on most commercial aircraft that enter U.S. airspace. In this chapter, we report on a case study of analyzing preliminary versions of its system requirements specification (SRS) using symbolic model checking. Although the versions had gone through certain independent verification and validation efforts, several anomalies were discovered using model checking. Preliminary results in this chapter appeared in Anderson et al. [9] and Chan et al. [34].

4.1 General Description

The TCAS-equipped aircraft is surrounded by a protected volume of airspace. When another TCAS-equipped aircraft intrudes into this volume, TCAS II generates warnings (traffic advisories) and suggests possible escape maneuvers (resolution advisories, or RAs) in the vertical direction to the pilot. Examples of RAs include Climb, Descend, Increase-Climb (“increase the current climb rate”), Increase-Decend, Climb-VSL0 (“do not descend”), and Climb-VSL500 (“do not descend more than 500 ft/min”). The complexity of the system stems from the vast number of inputs from the pilot, altimeter readings, and ground stations; the projections of flight paths based on current altitudes, velocities, and directions; the complicated logic for deriving RAs to maintain a safe separation while minimizing disruption; and the needs for avoiding false alarms and handling multiple aircraft, etc. It was described by the head of FAA as “the most complex system to be incorporated into the avionics of commercial aircraft” [107, p. 685].

The system used to lack a high-level requirements specification; the official specification was some pseudo-code. Leveson et al. [107] reverse-engineered the pseudo-code to obtain an RSML model, which was later adopted by the FAA as the official TCAS II SRS. Our analyses were based on two preliminary versions of the SRS, mainly version 6.00 dated March 1993. As far as we know, the problems we identified here are not present in the current version 7 of the document (DO-185A) [124], which can be obtained from the Radio Technical Commission for Aeronautics.
4.2 Obstacles to Model Checking

After we derived the translation rules in the previous chapter, we had to overcome a number of obstacles to make the model checking of the specification feasible.

4.2.1 TCAS II

The RSML document has 400 pages, and the first obstacle to analysis was its sheer size. We decided to try to verify a portion of it, namely a state machine called Own-Aircraft, which occupies about 30% of the specification. Figures 4.1 and 4.2 shows its state-machine diagrams. Own-Aircraft tracks the states of the TCAS-equipped aircraft, and has close interactions with another state machine called Other-Aircraft, which tracks the state of other aircraft in the vicinity and possibly generates RAs. Up to thirty other aircraft can be tracked. From the RAs given by all the instances of Other-Aircraft, Own-Aircraft derives a “composite RA” (a compromise among the RAs) and generates visual and audio outputs to the pilot. We note that most of the complexity of the specification lies in the guarding conditions, not in the state-machine diagrams. Figure 4.3 shows an example of a guarding condition in Own-Aircraft. The AND/OR tables denoting the guarding conditions are often large. The atomic predicates may refer to other macros defined by AND/OR tables given
Figure 4.2: Advisory-Status (part of Own-Aircraft) in TCAS II SRS

**Transition(s):** Yes → No

**Location:** Advisory-Status ➔ Corrective-Descend

**Trigger Event:** Corrective-Climb-Evaluated-Event

**Condition:**

<table>
<thead>
<tr>
<th>Condition</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don’t-Climb-Don’t-Descend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Climb-RA-Weakened</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Increase-RA-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Corrective-Climb in state <strong>No</strong></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Own-Track-Alt-Rate &lt; -500 ft/min</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Descend-RA-Weakened</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Own-Track-Alt-Rate ≤ Descend-Goal</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

**Output Action:** Corrective-Descend-Evaluated-Event

Figure 4.3: One of the state transitions in Own-Aircraft
elsewhere (e.g., Dont-Climb-Dont-Descend and Climb-RA-Weakened in Figure 4.3), and some of these predicates are arithmetic constraints over altitudes and altitude rates.

Since most of Own-Aircraft is supposed to be deterministic, we modeled it mainly based on the translation rules in Section 3.3.3, with the abstraction discussed below. We also created variables for any states of Other-Aircraft that are referenced within Own-Aircraft, and allowed nondeterministic transitions among the states using the translation explained in the last part of Section 3.3.4. We focused on resolution maneuvers with one intruder aircraft and thus modeled only one instance of Other-Aircraft.

4.2.2 BDDs

In addition to Boolean and enumerated variables, inputs to the system also include numbers, such as altitude and altitude rates. Different versions of the specification are inconsistent as to whether these numeric variables are integers or reals. Moreover, the ranges of some of them are not specified. To use BDDs, we had to assume that these inputs are bounded integers. Take altitudes for example. Some altitude variables are specified to have granularity as fine as “1 to 10 feet”, and are compared to constants ranging from 400 feet to 30 500 feet. Therefore at least 13 to 15 bits are needed to represent them.

Numeric inputs are referenced in guarding conditions, macros, or functions. Although BDDs under a suitable variable order can efficiently represent equality and inequality between linear expressions (e.g., $2alt_1 + 3alt_2 > alt_3$), there is provably no efficient BDD representation for multiplication or division of variables (e.g., $alt \times time > distance$) under any variable order [23, 133]. So, we needed to avoid them. Two functions in Own-Aircraft do involve multiplication and division of values for measured altitudes and altitude rates. These are measurements of input variables that we already modeled nondeterministically. So we made the abstraction to treat the calculated values as nondeterministic themselves. (We also eliminated from our model several input variables that are only referenced by the two functions.) The abstraction did not cause problems for the properties that we checked and report in Section 4.3.

4.2.3 SMV

**Linear Arithmetic**  The performance of BDD-based algorithms is directly related to the BDD size. Some of our early attempts at checking generated enormous BDDs: At one point the BDDs consumed 200 MB of physical memory, and other runs were terminated before the BDD was constructed. Our attempts to check formulas with the large BDDs were generally unsuccessful or too slow (our initial success in identifying nondeterminism discussed in Section 4.3.1 was an overnight run, which has been reduced to a few minutes).

The BDD size can be reduced by dynamic variable reordering and conjunctive partitioning [27], which are supported by the version of SMV that we used (Release 2.4.4). These techniques dramatically improved the performance of checking some formulas; however, they did not solve all the problems. The BDD size was very sensitive to the ranges of the variables representing altitudes and altitude rates. In fact, SMV cannot even efficiently handle our simple example program listed in Appendix A.1.
Initially we got around the problem by redefining the constants and reducing the variables to small ranges, for example, from 0 to 15 for altitudes and $-4$ to 3 for altitude rates. (Increasing the variables by one bit sometimes exploded the checking time from ten minutes to more than ten hours.) Although we were able to build the BDDs in this way and check some formulas, this \textit{ad hoc} solution was unsatisfactory in many ways. An obvious drawback is that because of the small ranges, some distinct constants in the specification became identical after the mapping (for example, both 400 feet and 1000 feet might become 1). This changed the behaviors of the model and caused invalid analysis results.

We could not leave the results of addition and comparison nondeterministic as we did with multiplication and division, because addition and comparison are essential to the logic of Own-Aircraft. For example, any Descend RA is prohibited when the difference between the current altitude of the own aircraft and the estimated ground level altitude is less than some threshold. If the subtraction or the comparison were modeled nondeterministically, this safety requirement would be violated trivially.

The problem with the ranges was due to SMV's inefficient implementation rather than the limitations of BDDs. As Yang et al. [139] observe, SMV is extremely inefficient in constructing BDDs for integers: Building the BDD for even a simple assignment like

$$\texttt{next}(x) := x;$$

requires time and space exponential in the number of bits of $x$. For expressions involving multiple variables, an additional problem is the variable ordering. For example, for two $n$-bit integers $X = x_{n-1}x_{n-2}\ldots x_0$ and $Y = y_{n-1}y_{n-2}\ldots y_0$, the BDD for $X = Y$ has size linear in $n$ if the variable order is $x_{n-1}, y_{n-1}, x_{n-2}, y_{n-2}, \ldots, x_0, y_0$, but requires exponential size if the order is $x_{n-1}, x_{n-2}, \ldots, x_0, y_{n-1}, y_{n-2}, \ldots, y_0$. If $X$ and $Y$ are declared in SMV with the code

\begin{verbatim}
VAR
  x: 0..N;
  y: 0..N;
\end{verbatim}

where $N = 2^n - 1$, SMV never interleaves the bits of $X$ and $Y$ in the BDD variable order and thus produces exponential-size BDDs for the predicate $X = Y$.

We considered two ways of attacking this problem, namely changing the internals of SMV, or doing addition and comparison at the source code level. Although in principle the former may be a better long term solution (Yang et al. [139] give an efficient algorithm for constructing BDDs for linear arithmetic predicates), the latter method seemed a simpler approach and we were able to use it with great success. We wrote some simple \texttt{awk} scripts to automatically generate the code

\begin{verbatim}
VAR
  x1: boolean;  y1: boolean;
  \vdots \vdots
  xn: boolean;  yn: boolean;
\end{verbatim}
to declare $X$ and $Y$, and $x_1 y_1 \& x_2 y_2 \& \ldots \& x_n y_n$ to represent the equality $X = Y$. Addition, subtraction, and inequality can be similarly translated. We can now model the altitudes and altitude rates with the precisions required by the specification. Changing the variables for altitudes from 4 bits to 15 bits and those for altitude rates from 3 bits to 13 bits blows up the size of the state space roughly from $10^{40}$ to $10^{65}$. However, this increase in precision increased the run time and the number of BDD nodes used by less than a factor of three. We also wrote an axk back-end to SMV to convert the bits back to integers for easy interpretation of the counterexamples.

**Counterexample Search** Counterexamples also presented performance problems. Generating a counterexample often took hours even though the formula was determined false within minutes. In Section 6.4, we will explain how we modified the model checker to speed up the counterexample search.

### 4.3 Results

Once we overcame these obstacles, we were ready to do some analysis of the specification using the model checker. We modeled the global state space with 227 Boolean variables, 10 of which are for events, 36 for the states of Own-Aircraft, 19 for the states of Other-Aircraft, 134 for altitude and altitude rates, 22 for inputs other than altitude and altitude rates, and 6 for other purposes. The size of the state space is about $1.4 \times 10^{65}$. The size of the *reachable* state space is at least $9.6 \times 10^{60}$. We obtained this lower bound by executing SMV with the command line option `-f` but without running it to completion. This option forces SMV to find the reachable state space before evaluating any formula.

The properties that we analyzed include general properties that should hold in most RSML specifications (Sections 4.3.1–4.3.3) and domain-specific properties (Sections 4.3.4 and 4.3.6). The violation of some of the properties was unknown to us before the analysis (Sections 4.3.2, 4.3.5, and 4.3.6). In Chapters 6 and 7 we will look at the performance of analyzing these properties in details. For now, it suffices to point out that each property could be analyzed in several minutes of CPU time.

Given an arbitrary system, it is often not obvious what domain-specific properties to verify. In our experiments, we based these properties on published documents and our own knowledge of the system. In this section, we only report the most interesting results we found.

**4.3.1 Transition Consistency**

We need to distinguish two kinds of nondeterministic transitions, namely those that are intentional, resulting from either the logic of the original specification (the rare case) or the abstraction that we employed (Section 4.2), and those that exist in the original specification but are unintentional, which we want to detect.

There are two reasons why we want to find the latter transitions. First, as Jaffe et al. [101] argue, nondeterminism in software requirements usually reflects *inconsistency* and should be avoided. Second, in our translation into SMV, we assumed the transitions are
Case Study: TCAS II

deterministic and separately dealt with the nondeterministic transitions as special cases (Section 3.3.4). If unintentional nondeterministic transitions are present, the SMV program in general will behave differently and all analyses will become invalid.

There are known nondeterministic transitions in early versions of the specification. For example, TCAS II has the notion of sensitivity level, which determines the volume of protected airspace around the aircraft. Some of the nondeterministic transitions allow a choice, under identical conditions, of increasing or decreasing the sensitivity level, which is clearly an inconsistency in these early versions of the specification. So, our first attempt was to find such transitions with the model checker. (For the other properties that we checked, we worked with a later draft specification, in which there are no inconsistencies in Own-Aircraft.) These nondeterministic transitions had previously been identified by Heimdahl and Leveson using a different technique [77]. We were interested in checking these properties to show that model checking could match previous results. In Section 4.4.1 we will summarize the differences between our model checking approach and their technique.

In our example in Figure 3.1 on page 14, transitions $t_9$ and $t_{12}$ can be enabled simultaneously. We can check this with the model checker by the CTL formula

$$\text{AG} \neg (t_9 \land t_{12}),$$

which says that the two transitions are never enabled simultaneously. We can check a similar formula for every pair of conflicting transitions that are not meant to be simultaneously enabled. This may seem a large number of cases to check, but often, the guarding conditions alone prevent the transitions from being enabled at the same time, such as the transitions in Alt-Layer. (Indeed, this is the premise of Heimdahl and Leveson’s technique.) In this case, the state space is not explored, because the BDD quickly reduces the CTL formula to $\text{AG} \text{true}$, which in turn is trivially evaluated to true. Otherwise, the model checker will search for a counterexample. Using this technique, we were able to find the nondeterministic transitions in a version of the TCAS II specification, and verify that these transitions do not exist in a later version. The particular formula that reveals the inconsistency is

$$\text{AG} \bigvee_{t_1 \neq t_2 \land \text{scope}(t_1) = \text{scope}(t_2) = \text{Auto-SL}} t_1 \land t_2.$$  \hfill (4.1)

A subtle but important issue demands additional attention: As mentioned, our translation is not faithful if the RSML machines contain unintentional nondeterministic transitions. So, it may seem circular to show the absence of such transitions in the RSML machines using the translated SMV program.

However, we can prove that this is not a problem.

**Lemma 4.1** Given two state transition systems $M_1 = \langle Q, R_1, I \rangle$ and $M_2 = \langle Q, R_2, I \rangle$ with identical state spaces and initial states. Define

$$N = \{ q \in Q \mid \exists q', (q, q') \in (R_2 - R_1) \cup (R_1 - R_2) \}.$$

The set $N$ is reachable in $M_1$ if and only if it is reachable in $M_2$.

The set $N$ in the lemma is the set of “bad” global states that may lead to different behaviors in $M_1$ and $M_2$. The lemma says that some state in $N$ is reachable in $M_1$ if and
only if some state in \( N \) is reachable in \( M_2 \), but does not require that the two states be the same. Intuitively, this is true because the shortest path to any state of \( N \) in \( M_1 \) must also appear in \( M_2 \) and vice versa.

**Proof:** We assume that \( N \) is reachable in \( M_2 \) and argue that it is reachable in \( M_1 \) as well. The other direction is symmetric.

By the definition of reachability, there exists a finite sequence of states \( q_0, q_1, \ldots, q_n \) such that \( q_0 \in I \), \( q_n \in N \) and

\[
(q_i, q_{i+1}) \in R_2 \text{ for every } i < n.
\]

(4.2)

Let \( k \) be the smallest \( i \) with \( q_i \in N \); that is, \( q_k \in N \) and

\[
q_j \notin N \text{ for every } j < k.
\]

(4.3)

We argue \( q_k \) is reachable in \( M_1 \) by showing that \( q_0, q_1, \ldots, q_k \) is a prefix of some path in \( M_1 \). Since by assumption \( q_0 \in I \), we only need to show \( (q_j, q_{j+1}) \in R_1 \) for every \( j < k \). For every such \( j \), from (4.2) we know that \( (q_j, q_{j+1}) \in R_2 \). If \( (q_j, q_{j+1}) \notin R_1 \), then by the definition of \( N \), we will have \( q_j \notin N \), contradicting (4.3). Therefore, we must have \( (q_j, q_{j+1}) \in R_1 \). \( \square \)

Let \( M_{RSML} \) and \( M_{SMV} \) be the state transition systems representing the RSML machine and the translated SMV program respectively, and let \( N \) be as defined above with \( M_1 \) and \( M_2 \) being \( M_{RSML} \) and \( M_{SMV} \). Let's assume for now that abstraction by nondeterminism discussed in Section 4.2 was not used. By definition, the set \( N \) contains precisely the set of global states that are not faithfully translated. Because our translation handles deterministic transitions (and the intentional nondeterministic transitions) faithfully, the set \( N \) is exactly the set of global states that contains unintentional nondeterministic transitions. This means that \( N \) is reachable in \( M_{RSML} \) if and only if \( M_{RSML} \) exhibits unintentional nondeterministic behavior. Therefore, by the lemma, it is sufficient to analyze \( M_{SMV} \) to detect the nondeterminism, and, in addition, we will not obtain false negative results. If, on the other hand, some intentionally nondeterministic transitions of \( M_{RSML} \) are mistakenly modeled as deterministic ones in \( M_{SMV} \), this analysis will reveal them and the designer can then use this information to correct the model.

False negatives are in principle possible when abstraction is used, because the set \( N \) may now be reachable in \( M_{SMV} \) but not in \( M_{RSML} \). However, such false negatives did not happen when we analyzed nondeterminism in our experiments. We did find false negatives when checking other properties as discussed below.

**4.3.2 Function Consistency**

The value of the function Displayed-Model-Goal, shown in Figure 4.4, is displayed to the pilot when an event called Composite-RA-Evaluated-Event occurs. (Most of the identifiers in the figure are RSML macros or abbreviations, the definitions of which are omitted here.) The function represents the optimal altitude rate at which the pilot should aim (a positive value indicates the upward direction). The function definition consists of eight cases, which are supposed to be mutually exclusive. It is not obvious whether this is the case since the mutual exclusion depends on logic elsewhere in the specification.
Case Study: TCAS II

\[
\text{Displayed-Model-Goal} = \begin{cases} 
0 & \text{if Composite-RA not in state Positive} \\
\text{Max}(\text{Own-Track Alt-Rate}, & \text{if (New-Climb or New-Threat) and} \\
\text{Prev (Displayed-Model-Goal),} & \text{not New-Increase-Climb and} \\
1500 \text{ ft/min}) & \text{not (Increase-Climb-Cancelled or} \\
& \text{Increase-Descend-Cancelled) and} \\
& \text{Composite-RA in state Climb} \\
\text{Min}(\text{Own-Track Alt-Rate,} & \text{if (New-Descend or New-Threat) and} \\
\text{Prev (Displayed-Model-Goal),} & \text{not New-Increase-Descend and} \\
-1500 \text{ ft/min}) & \text{not (Increase-Climb-Cancelled or} \\
& \text{Increase-Descend-Cancelled) and} \\
& \text{Composite-RA in state Descend} \\
2500 \text{ ft/min} & \text{if New-Increase-Climb} \\
-2500 \text{ ft/min} & \text{if New-Increase-Descend} \\
\text{Max}(\text{Own-Track Alt-Rate,} & \text{if Increase-Climb-Cancelled and} \\
1500 \text{ ft/min}) & \text{not New-Increase-Climb and} \\
& \text{Composite-RA in state Positive} \\
\text{Min}(\text{Own-Track Alt-Rate,} & \text{if Increase-Descend-Cancelled and} \\
-1500 \text{ ft/min}) & \text{not New-Increase-Descend and} \\
& \text{Composite-RA in state Positive} \\
\text{Prev (Displayed-Model-Goal)} & \text{Otherwise}
\end{cases} \\
\text{/* case 1 */} \quad \text{/* case 2 */} \quad \text{/* case 3 */} \quad \text{/* case 4 */} \quad \text{/* case 5 */} \quad \text{/* case 6 */} \quad \text{/* case 7 */} \quad \text{/* case 8 */}
\]

Figure 4.4: Definition of Displayed-Model-Goal

Checking for mutual exclusion of the cases, which we call function consistency, is similar to checking for transition consistency. We defined a Boolean symbol \text{Case-}i \text{ for the } i^{th} \text{ case, and checked the CTL formula}

\[
\forall i \exists j (\text{Case-}i \land \text{Case-}j) = \text{false}.
\]

The model checker found a counterexample showing that the formula was false. After carefully examining the counterexample, we decided that the scenario was due to the oversimplified model of Other-Aircraft, which we had considered as a part of the nondeterministic environment. In the counterexample, Other-Aircraft reverses from an Increase-Climb RA to an Increase-Descend RA in one step, which is prohibited by the logic in the specification. After we changed the code to prevent Other-Aircraft from making such spurious transitions, no counterexamples were found.

This refinement of Other-Aircraft to allow successful checking of a property has implications for the use of model checking during the development of specifications. In essence, the examination of the scenario and the subsequent refinement can be considered to be a way of documenting an intended, but implicit, interaction between the Own-Aircraft and Other-Aircraft state machines. As an after-the-fact occurrence, as in our case, the refinement is an effective way to allow us to translate and check properties on a portion of the specification, rather than on the full specification. If done as the specification was developed, it could
also be an effective way to understand and document the interactions between the parts of the specification.

4.3.3 Macrostep Termination

A macrostep in an RSML state machine may not terminate if the machine contains a cycle of events under the transition relation \[107\]. However, the precedence relation of the events in an RSML specification is usually acyclic (see Section 6.1.2), so it is easy to see that a step will always terminate; this happens in the TCAS II specification. Alternatively, we can verify termination with the CTL formula

\[
AGAF \text{ stable}
\]

which means that the system is stable infinitely often. In other words, it can only stay unstable for a finite number of microsteps. This formula was found to be true, as expected.

4.3.4 Inhibition of Resolution Advisories

A TCAS II document \[60\] claims that (1) all Descend RAs are inhibited when the own aircraft is below 1000 feet above ground level, and (2) all Increase-Descend RAs are inhibited below 1450 feet above ground level. The logic that guarantees these safety properties resides in both Own-Aircraft and Other-Aircraft. We imposed the necessary constraints on the transitions of Other-Aircraft in order to check whether the part of the logic in Own-Aircraft is correct. The model checker found that while the first property is satisfied, the second is not. The formula that we checked for the second property was roughly

\[
AG ((\text{stable} \land \text{Radio-Altimeter-Status} = \text{Valid} \land \text{Own-Alt-Radio} \leq 1450) \\
\rightarrow \neg \text{Increase-Descend})
\]

where Own-Alt-Radio is an input representing the altitude of the own aircraft above ground level, Radio-Altimeter-Status an input indicating whether Own-Alt-Radio is valid, and Increase-Descend an expression evaluating to true when an Increase-Descend RA is issued. (As mentioned in Section 4.2.3, the inequality is actually a long expression relating the bits of Own-Alt-Radio.) The counterexample given by the model checker revealed a typographical error in a guarding condition in the specification (\(>\) instead of \(\leq\)). (We discovered the typographical error by observation during the translation process.) The effect of the error was that the Increase-Descend RA was inhibited for at most one step, thus allowing the safety property to be violated.

4.3.5 Output Agreement

When Composite-RA-Evaluated-Event occurs, in addition to the value of Displayed-Model-Goal, the state of Composite-RA in Figure 3.5 on page 24 (shown also in Figure 4.2 on page 35) is also given to the pilot. Therefore it seems safety-critical that Composite-RA and Displayed-Model-Goal agree with each other. We checked for several such properties. For example, one would expect that if Composite-RA is in state Climb, then Displayed-Model-Goal should be at least 1500 ft/min. However, the model checker revealed that this
is not true. In fact, it showed a stronger result: When Composite-RA is Climb, Displayed-
Model-Goal could be negative. The CTL formula we checked was

\[ AG((\text{Composite-RA} = \text{Climb}) \land \text{Composite-RA-Evaluated-Event}) \]
\[ \rightarrow \text{Displayed-Model-Goal} \geq 1500 \]  
\[ (4.7) \]

The counterexample given by the model checker was a three-step scenario (consisting of
23 global transitions):

1. At time \( t_0 \), there is an intruder aircraft and Other-Aircraft gives a Descend RA. As
a result, Composite-RA is in state Descend and by case 3 in Figure 4.4, we have
Displayed-Model-Goal \( \leq -1500 \) ft/min.

2. At time \( t_1 > t_0 \), Other-Aircraft realizes that an increase in descend rate is neces-
sary and issues an Increase-Descend RA, which puts Displayed-Model-Goal at \(-2500 \)
ft/min by case 5.

3. At time \( t_1 + 1 \), the situation has changed and Other-Aircraft projects that a climb
would result in greater separation from the intruder. So it reverses its RA to Climb,
making Composite-RA enter state Climb. At that point, case 7 applies, and Displayed-
Model-Goal becomes less than \(-1500 \) ft/min, resulting in contradictory outputs.

Another output-agreement property is that when a new Increase-Climb RA is issued,
the value of Displayed-Model-Goal should not decrease. The result was similar to that of
function consistency: The model checker found a counterexample, which was due to the
overly abstract model of Other-Aircraft. After refining the model, no counterexamples were
found. The CTL formula checked was

\[ AG((\text{Composite-RA-Evaluated-Event} \land \text{New-Increase-Climb}) \]
\[ \rightarrow \text{Displayed-Model-Goal} = \text{prev-Displayed-Model-Goal}) \].  
\[ (4.8) \]

4.3.6 Mutual Exclusion of Corrective Climb and Descent

Britt [20, p. 49] states that Own-Aircraft should never be in two local states Corrective-
Climb\( \triangleright \) Yes and Corrective-Descend\( \triangleright \) Yes simultaneously. Comments in our version of the
specification, however, explicitly say that the two local states are not mutually exclusive.
So we checked the property

\[ AG((\text{Corrective-Climb-Evaluated-Event} \]
\[ \rightarrow \lnot((\text{Corrective-Climb} = \text{Yes} \land \text{Corrective-Descend} = \text{Yes})) \).  
\[ (4.9) \]

The model checker found a counterexample to the property, confirming the comments in
the requirements. We note that this property was particularly hard to analyze, and was
feasible to check only after the optimization to be presented in Section 6.1 was used.
4.3.7 Miscellaneous

The value of any \( \text{Prev}(\text{expr}) \) is undefined in the first step. As mentioned in Section 3.3.6, we did not constrain the initial value of the SMV variable representing \( \text{Prev}(\text{expr}) \) to let the model checker find an initial value that falsifies the property being checked. So while verifying the properties mentioned above, we also discovered situations in which \( \text{Prev} \) values are referenced in the first step.

In addition to AG and AGAF formulas, we also checked some formulas of the form AGEF \( p \), which asserts that \( p \) is always possible in the future. For example, \( p \) may be a predicate on inputs, as in the following formulas:

\[
\text{AGEF } \text{Radio-Altimeter-Status} = \text{Valid} \quad (4.10)
\]
\[
\text{AGEF } \text{Radio-Altimeter-Status} = \text{Invalid.} \quad (4.11)
\]

Note, however, that verifying such formulas does not establish any property of the RSML specification; it is merely a sanity check to ensure that our model does not prevent the environment from changing.

Another common use of AGEF formulas is to specify that it is always possible to shut down or restart the system. While the notions of shutdown or restart are not applicable to our model of Own-Aircraft, we could check, for example, that it is always possible for the system to enter certain states or produce certain outputs (for example, the system is never locked in a certain RA that no inputs can change). However, because nondeterminism was used to abstract out certain details, a behavior that is possible in the SMV program is not guaranteed to exist in the RSML machine. So the analysis of such AGEF formulas is not sound. (This problem can be solved by a recent technique called module checking [104].)

4.4 Discussion

In this section, we discuss related work and argue for using model checkers as design tools.

4.4.1 Related Work

Consistency and Completeness Checking The first automatic verification work done on the SRS was by Heimdahl and Leveson [77]. Instead of exploring the state space, they compose results of local analyses to deduce global properties. However, the properties that we checked were different. Their concerns were transition consistency and completeness [101], which are domain-independent properties. In Section 4.3.1 we discussed how we verified transition consistency. Completeness intuitively means that a response is specified for every input, and in principle can also be checked in our framework. In general, our approach permits analysis of arbitrary CTL formulas, and is therefore capable of verifying domain-specific properties as well.

Consider consistency in more detail. Their tool checks that the conjunction of the guarding conditions of every pair of conflicting transitions with the same trigger is a contradiction. That is, for the example in Figure 3.1 on page 14, while we check whether the CTL formula \( \text{AG} \neg (t_1 \land t_5) \) holds in the system, they check whether the conjunction of the guarding
conditions of \( t_1 \) and \( t_5 \) is a contradiction. In general, their method can be less accurate, for three orthogonal reasons.

First of all, since they did not explore the reachable state space, the states that exhibit inconsistency or incompleteness may not be reachable. In other words, when the conjunction of the guarding conditions is satisfiable, the user is responsible for determining whether the failure represents a genuine problem, while in our case, the model checker will help by finding a counterexample. This is an inherent limitation of their approach, but is also an inherent advantage, because it allows simple analysis.

The second source of inaccuracy stems from their decision to consider only transitions with the same trigger. Consider again Figure 3.1. Conflicting transitions \( t_0 \) and \( t_{12} \) can be simultaneously enabled because their triggers \( u \) and \( v \) may occur at the same time. Their tool, however, would fail to detect them, simply because it never considers transitions with different triggers together. (On the other hand, if it makes the conservative assumption that any subset of events may occur at the same time, it will mistakenly report that \( t_0 \) and \( t_{30} \) may cause nondeterminism, without realizing that their triggers \( u \) and \( w \) are mutually exclusive in any reachable states.)

The last source of inaccuracy in their method is the way they construct a Boolean formula for checking: They create one Boolean variable for each predicate in the guarding condition. For example, to check whether \( t_4 \) and \( t_7 \) are mutually exclusive, they would have a Boolean variable \( x_1 \) for \( alt > 10050 \) and another variable \( x_2 \) for \( alt < 1950 \), and check whether \( x_1 \land x_2 \) is a contradiction. This clearly results in a false negative. Heimdahl and Cerny [75] use a theorem prover to attack this problem. On the other hand, for the same property, we would have 15 Boolean variables representing the bit encoding of \( alt \) and then construct the BDD for the predicate \( alt > 10050 \land alt < 1950 \), which is automatically reduced to a contradiction. Note that although we would have more Boolean variables, the BDD size scales well for inequalities and linear arithmetic operations. A disadvantage is the inability to deal with real numbers, which have to be discretized as bounded integers. BDDs also cannot efficiently handle the complicated nonlinear predicates in TCAS II, but currently neither can their theorem-proving approach.

Although model checking can give more accurate results, it is also more costly. However, the two approaches are complementary and can be used together for system development or verification.

**Hybrid Systems** Our verification results are robust in the sense that, except for the synchrony hypothesis, we do not make any assumption about the environment, which includes the pilot and the aircraft. However, we cannot verify properties that depend on the environment, such as “two aircraft will not collide if the pilots follow the RAs.” In addition to robustness, the reason for not having modeled the environment more precisely is the lack of such information in the specification. In principle, were such information available, we could have incorporated it in our model, but we would have to discretize the inherently continuous environment.

Verification of hybrid systems tackles this problem by modeling the environment with a set of real-valued variables governed by constraints on their derivatives [4]. The complexity of model checking becomes much higher and some problems even become undecidable, but
symbolic model checkers for hybrid systems have been built [83]. However, the models analyzed in published case studies are orders-of-magnitude smaller than the examples we studied. It would be interesting to see whether the next-generation tools can scale to significantly larger systems.

**Miscellaneous** Lygeros and Lynch [110] develop a methodology for verifying the conflict-resolution algorithm in TCAS, but the method is not automatic and is not based on the RSML specification.

### 4.4.2 Model Checking as a Design Tool

Although we used model checking to analyze requirements after they were fully specified, we believe that the greatest benefits of model checking come as a tool for developing the specification. In this section we give some evidence that the approach can be valuable.

**Understanding and Documentation** As shown in Section 4.3.2, we sometimes obtained false counterexamples because of the over-abstracted model. Only when we refined the model to remove the spurious transitions could we verify the properties in question. This process of getting incorrect counterexamples and then removing them may seem counterproductive, but there are a number of reasons why this approach is in fact useful. A software engineer can use the information obtained from analyzing the counterexamples to clarify the relationship between parts of the specification, in particular between those parts that are fully modeled and those that are partially modeled. In complex specifications like TCAS II, the interconnections between the subsystems are often not fully described and documented. Our style of model checking can be viewed as a way of learning about and documenting the interconnections between parts of the specification.

**Iterative Development** Furthermore, we claim that this iterative approach can serve as a development tool. A common conception is that verification is the finale of the specification process—it either shows correctness or reveals problems to be fixed. This view makes verification less effective in two ways. First, the complete specification may be too large to analyze, so abstraction becomes necessary to cope with the complexity. Second, when problems are found, fixing them is expensive this late in the specification stage (although still less costly than problems found in the implementation).

Using verification techniques early in the development cycle to interleave design and analysis can tackle these problems. The complexity only gradually increases as the specification evolves, and verification at early stages is more likely to be tractable. In addition, analysis results can give fast feedback to designers to improve the cost-effectiveness of the technique. Researchers on hardware verification have also pointed out some advantages of early use of verification [112].

For example, when developing the TCAS II specification, an engineer could have specified Own-Aircraft first and have left Other-Aircraft nondeterministic. Then an analyst could have analyzed Own-Aircraft with model checking and discovered the assumptions on the behaviors of Other-Aircraft that are necessary for Own-Aircraft’s correct operations. This
information then could have been used to develop Other-Aircraft. During the development of Other-Aircraft, the properties can be re-checked, as with regression testing, to ensure that the properties are continually maintained.
Case Study: EPD System

In our second case study, in collaboration with William Warner and David Jones of Boeing, we analyzed their statecharts model of the electrical power distribution (EPD) system on Boeing aircraft. Several faults of the model were uncovered, although we were quite confident of its correctness based on simulation results. We also discuss how model checking could potentially be used to benefit the model-based development processes used at Boeing. Preliminary results appeared in Chan et al. [35].

5.1 General Description

The purpose of the EPD system is to distribute AC and DC power to other airplane systems. It comprises separate interconnected distribution systems including main AC power, backup AC power, DC power, standby power, and flight controls power. Electrical power is distributed from power sources to power buses via a number of relayed circuit breakers. Failures of the power sources or circuit breakers are automatically detected and isolated.

Figure 5.1(a) depicts part of the system configuration in normal operations. The power buses $l_{\text{main}}$ and $r_{\text{main}}$ belong to the main AC power subsystem, and are normally powered by the generators $l_{\text{gen}}$ and $r_{\text{gen}}$ respectively. When $l_{\text{gen}}$ loses its power because of either manual shutdown or failure, the circuit breakers will be reconfigured automatically to use $r_{\text{gen}}$ to power both $l_{\text{main}}$ and $r_{\text{main}}$, as illustrated in Figure 5.1(b). The same

![Diagram of power source and circuit breaker configurations](image)

(a) Normal operations  
(b) $l_{\text{gen}}$ fails

Figure 5.1: Handling a power-source failure in the EPD model
configuration may also result from failures in the circuit breakers that connect \( l_{\text{gen}} \) and \( l_{\text{main}} \). The system is supposed to satisfy a number of stringent requirements, such as the resilience of the power buses against single or multiple failures in the power sources and/or the circuit breakers.

A circuit breaker, either open or closed at any moment, is modeled as a two-state machine and is managed by a controller. Figure 5.2 shows a generic circuit breaker and its controller. The transitions in the circuit-breaker state machine are guarded by the complement of a Boolean input \( f \) that indicates a failure, so a failed circuit breaker does not respond to the controller. The guarding condition \( c \) of the controller is usually a nontrivial predicate relating inputs and the local states of other circuit breakers and the power sources.

We stress that the statecharts model was developed for research purposes and does not represent the actual requirements used to develop the on-board system. As such, the model by intent did not include all the logic necessary for a complete specification. The model was intended as a high-level abstraction of the electrical system, which included only the logic necessary to accomplish the goals of a wider airplane system analysis [116].

5.2 Obstacles to Model Checking

We focus on the portion of the statecharts that models the main and backup AC distribution subsystems; other subsystems were abstracted away manually. There are 33 two-state machines, 23 Boolean inputs, and 34 events. With 11 Boolean state variables for other purposes, there are altogether 101 Boolean state variables, or about \( 10^{27} \) global states, of which at least \( 10^{15} \) are reachable. The major obstacle in the model-checking analysis was efficiency: After we translated the statecharts model to SMV, we quickly ran into performance problem. We were able to obtain the results below only after using our optimization techniques presented in Chapters 6 and 7.

5.3 Results

The analyses can be divided into analyses on normal behaviors (i.e., no component failures) and fault tolerance (single and multiple failures). We report some of the more interesting
Case Study: EPD System

results here. Although the model had been exercised extensively in simulation, several flaws were discovered using model checking.

Normal Operations

In normal operations, all buses in the main and backup AC subsystems should be powered in the stable states. We checked the formula

\[ AG((stable \land no-failures) \rightarrow (main \land backup)) \]  

(5.1)

where no-failures is a proposition indicating the absence of failures (each of the 17 failures is represented by an atomic proposition), and main and backup assert respectively that the main buses (l_main and r_main) and backup buses are powered. Note that the formula does not simply ignore failures; it takes into account scenarios in which failures occur but are subsequently recovered. The formula was evaluated true by the model checker.

Not only should the buses be powered when there are no failures, they should be powered by different sources. We checked the formula

\[ AG((stable \land no-failures) \rightarrow separate-sources) \]  

(5.2)

where the proposition separate-sources asserts that a power source is connected to at most one bus. This time, however, the model checker gave a counterexample revealing a bug in the model of the circuit breakers. In the counterexample, r_gen initially powers both l_main and r_main because of a failure in the circuit breakers. Assume the failed circuit breaker is modeled by the machine CB in Figure 5.2. The recovery of CB corresponds to the Boolean input f changing to false. This change alone, however, cannot trigger any local transition, as the transitions in CB are guarded by events. So when CB recovers, the system ends up in a situation in which there are no failures, but r_gen is still powering both main buses, violating the formula. We refer to this bug as B1, which we fixed by making CB go to the local state indicated by its controller upon recovery. With this bug fix, the formula was successfully verified.

Fault Tolerance

The main buses should in fact tolerate one failure in the power sources or circuit breakers. We checked the formula

\[ AG((stable \land at-most-1-failure) \rightarrow main) \]  

(5.3)

where the proposition at-most-1-failure has the obvious meaning. The model checker gave a counterexample that again reveals the bug B1, although the scenario is more complex. It involves a failure in a circuit breaker, a change in inputs to induce a state change in its controller, the circuit breaker’s recovery, and a subsequent failure in one of the power sources. After we fixed the bug and rechecked the formula, the model checker gave another counterexample that disclosed a logical flaw—one of the circuit breakers does not respond to a failure in another circuit breaker that it is supposed to handle, resulting in power loss.
to both main buses. We refer this bug as B2. (We have not attempted to fix this bug in this study.)

We initially thought that the backup buses should survive two failures. We checked this property, to which the model checker gave a counterexample with only one of the backup buses operating in the presence of two failures. After carefully examining the trace and studying the requirements document, we realized that the property actually is not supposed to hold—either one, but not necessarily both, of the backup buses should operate in that situation. We modified the formula accordingly:

$$AG((\text{stable} \land \text{at-most-2-failures}) \rightarrow \text{at-least-1-backup}).$$

(5.4)

The model checker responded with a counterexample exposing a logical flaw similar to B2 above. The counterexample involves simultaneous failures of two power sources, their subsequent recovery, and then simultaneous failures of two circuit breakers.

Miscellaneous

The formulas above are only concerned with stable states. One might expect certain causality to be maintained even in the unstable states. For example, the formulas do not prevent the power from going off within a macrostep before failures occur, as long as the right thing happens at the end of the macrostep. So, we evaluated formulas such as

$$AG(\text{main} \rightarrow A(\text{main} \land \text{no-failures}))$$

(5.5)

which asserts that, even in the unstable states, if the main buses are powered, then the power should persist unless a failure occurs. Interestingly, the model checker showed various scenarios violating such formulas—some situations that we do not regard as failures can cause transient power loss to the buses. Although this does not reflect any flaw in the system, it is still an interesting find, as the scenarios were not obvious to us before the analysis. Such results can provide insights into the design of the model and can reveal design flaws in some cases.

Other properties that we verified include the impossibility of having certain circuit breakers closed simultaneously (which would indicate some illegal system configuration), and other sanity checks, such as the property that if no power sources are operating, then no buses should be powered.

5.4 Discussion

A major goal of the case study was to evaluate the use of model checking as a debugger in support of requirements validation at Boeing by providing an additional debugging tool over and above the existing use of simulation. The use of modeling and simulation to support requirements validation at Boeing is described in Nobe and Bingle [115]. In this process, the written specification is developed first, and then a model is created to assist in validation of the requirements. Typically the model is simulated and executed by providing user-oriented inputs to the model and monitoring responses through panel graphics that represent actual system interfaces. Model checking could potentially help to ensure that the model reflects
other key design goals in that many of the system properties checked in this case study are not revealed in the operator interface.

Some flaws found during model checking might have been found if simulation runs had been explicitly defined to test conformance. However, the simulations would have had to include an extensive test suite, which included cases of intermittent failures of components to find the class of errors found during our model checking. Model checking appears to be particularly beneficial in helping find these “corner cases” with a minimum of additional effort.

The analysis described was done several years after the development of the model. However, it is clear to us that use of model checking during the initial development of the model would have detected subtle flaws before they were repeated throughout a much larger model. For example, the bug B1 repeats in every state machine that models a circuit breaker, and bugs similar to B2 appear in several places. In fact, some of these flaws could be found by focusing on the main AC subsystem and ignoring the backup AC subsystem.
Chapter 6

State-Set Optimizations

As mentioned in the previous two chapters, some of the analyses in the case studies were feasible only after we performed various optimizations. The specific techniques to be discussed in this and the next chapter are:

- Managing forward and backward traversals, to reduce the size of the BDD generated at each search iteration. Although state-space search can be done either forward or backward, we found that forward traversals are much less efficient for our models than backward ones (Section 6.4), and we further improved backward traversals by making certain invariants explicit in the search (Section 6.1).

- Semantics-preservation transformation of the model, to again reduce the size of the BDDs generated. We identified certain styles for synchronization in statecharts that are more efficient for symbolic model checking (Section 6.2). For statecharts not written in these styles, we developed procedures to automatically modify their internal representations to greatly improve the performance of their analysis. This was achieved by transparently incorporating a so-called microstep counter into the statecharts to take over the synchronization (Section 6.3).

- More sophisticated conjunctive partitioning of the transition relation and applying disjunctive partitioning in an unusual way, to reduce the size of the intermediate BDDs at each iteration. Further improvements were made by combining the two techniques to obtain DNF partitioning. (Section 7.1)

- Abstraction to decrease the number of BDD variables. Given a property to check, we can perform a simple dependency analysis to generate a reduced model that is guaranteed to give the same results as with the full model (Section 7.2).

- Short-circuiting to reduce the number of BDDs generated by stopping the iterations before a fixed point is reached (Section 7.3).

Experimental results will be provided showing how each of the techniques affected the performance of the analyses, sometimes by orders of magnitude. All our TCAS II experiments were performed on a Sun Sparc 10 with 128MB of main memory, while other data were collected on a Sun Ultra 2 with 256 MB of main memory.

In this chapter, we will first examine the techniques for reducing the BDD size representing state sets (the first two bullets above). Particularly worth highlighting is the technique
of using a microstep counter. The technique is intriguing, especially because it achieved substantial time and space improvements in our case studies even though the numbers of state variables, reachable states, and search iterations were all increased—exactly the opposite of what most existing techniques attempt to do to tame BDD blow-ups. Preliminary results in this chapter appeared in Chan et al. [35, 37].

6.1 Pruning Backward Traversals

Recall from Section 2.1 that, to evaluate the CTL formula $\text{AG}¬\text{error}$, we compute the fixed-point $\text{Pre}^*(E)$ backward from the set $E$ of error states, and see whether its intersection with the set of initial states is empty. A disadvantage of backward traversals is that they are likely to visit many unreachable, and thus irrelevant, states. However, we can prune a backward traversal if we know some upper bound on the reachable states. Notice that any invariant over the state variables describes a condition satisfied by every reachable state, and thus corresponds to such a bound. Some invariants, particularly those with small BDDs, can speed up backward traversals if they are incorporated into the search. In particular, the transitions in some statecharts cannot be taken at the same time, but this fact is lost in backward traversals. A specific invariant that we find useful to rectify this problem in TCAS II is the mutual exclusion of these transitions’ trigger events.

6.1.1 An Example with Mutually Exclusive Events

Consider the system in Figure 6.1. Event $x_0$ is the only external event. The conditions $c_1$ and $c_2$ are Boolean inputs, and the machines are initially in 0. When $x_0$ occurs, machine $A_1$ moves between states 0 and 1 depending on the condition $c_1$. If $A_1$ changes its state, in the next microstep machine $A_2$ may change its state depending on the condition $c_2$. There is no simultaneous concurrency in the system—at most one local transition can be enabled at any time. In particular, the transitions in $A_2$ can only be taken after those in $A_1$. (The example also demonstrates “non-oblivious synchronization”, which we will discuss later in Section 6.2.)

![Figure 6.1: State machine with mutually exclusive events and non-oblivious synchronization](image)
However, a backward traversal may consider many simultaneous transitions, which cannot occur in any execution. More explicitly, suppose we want to check whether the system can be in $A_1 \triangleright 1$ and $A_2 \triangleright 1$ simultaneously. Traversing backward, we find that one microstep before, the system may be in $(A_1 \triangleright 0, A_2 \triangleright 1)$, $(A_1 \triangleright 1, A_2 \triangleright 0)$, or $(A_1 \triangleright 0, A_2 \triangleright 0)$. The last case, however, is not possible, because events $x_0$ and $x_1$ cannot occur at the same time. (Notice that this is true only because we assume the synchrony hypothesis.) When there are more state machines and the guarding conditions are complex, such unnecessary exploration of concurrent transitions may cause BDD blow-up.

6.1.2 Pruning Using Invariants

Sometimes we can greatly simplify the search by observing that the events ($x_0$, $x_1$, and $x_2$ in our example) are mutually exclusive. This invariant can be incorporated into the traversals by simply conjoining it with the transition relation $R$. That is, if $\Sigma$ is the set of state variables encoding the mutually exclusive events, we can compute the conjunction of $R$ and

$$\wedge_{e_1,e_2 \in \Sigma} \neg(e_1 \land e_2)$$

and use the result as the transition relation in the traversals. This can be done in SMV by putting the invariant in a TRANS statement. (Or, we could use the invariant to simplify the state sets with a technique called “don’t-care minimization”. This can be achieved with the INVAR keyword in the recent versions of SMV.)

This technique requires finding a set of mutually exclusive events. To do this, we may perform a simple conservative static analysis on the precedence relation of the events. For now, we assume the absence of condition-driven transitions (Section 3.3.7), which will be addressed later in Section 6.3.4. We define $\prec$ to be the binary relation over the events, such that for each event $e_1$ and $e_2$, we have $e_1 \prec e_2$, or $e_1$ precedes $e_2$, if there exists a transition labeled with $e_1 | e_2$ for some guarding condition $e$. We assume that $\prec$ is acyclic, that is, $(e, e) \notin \prec^+$ for each $e$, where $\prec^+$ is the (non-reflexive) transitive closure of $\prec$. Many systems have this property because it prevents the nontermination of macrosteps, a design flaw that is potentially hard to locate.

For each event $e$, let $\sigma(e)$ be the smallest set of integers such that

1. $1 \in \sigma(e)$ if $e$ is an external event, and
2. for each $e$, if $i \in \sigma(e)$ then $i + 1 \in \sigma(e')$ for each $e'$ with $e \prec e'$.

Intuitively, $i$ is in $\sigma(e)$ if $e$ can occur just before the $i$th microstep of some macrostep. Since $\prec$ is acyclic, the set $\sigma(e)$ is finite, and the values of $\sigma(e)$ for all $e$ can be computed in time cubic in the number of events. Two events $e_1$ and $e_2$ are then mutually exclusive if the intersection of $\sigma(e_1)$ and $\sigma(e_2)$ is empty. For Figure 6.1, we have $x_0 \prec x_1 \prec x_2$, $\sigma(x_0) = \{1\}$, $\sigma(x_1) = \{2\}$, and $\sigma(x_2) = \{3\}$. So all the events are mutually exclusive.

As an alternative to performing this static analysis, the designer or analyst may know such a set of mutually exclusive events already, because the system’s synchronization structure may have been designed under careful consideration. This is indeed the case for the portion of TCAS II that we looked at: Its set of mutually exclusive events is evident.
Table 6.1 Performance of pruning for TCAS II. Time is in seconds and number of BDD nodes is in thousands. MX represents pruning using mutually exclusive events, and $\infty$ indicates timeout after one hour. Explained in Section 4.3, the properties, from left to right, are Increase-Descent Inhibition, Function Consistency, Transition Consistency, Output Agreement, and Mutual Exclusion of Corrective Climb and Descent. The order reflects the relative difficulty in analyzing them.

<table>
<thead>
<tr>
<th></th>
<th>(4.6)</th>
<th>(4.4)</th>
<th>(4.1)</th>
<th>(4.7)</th>
<th>(4.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
</tr>
<tr>
<td></td>
<td>(s) (K)</td>
<td>(s) (K)</td>
<td>(s) (K)</td>
<td>(s) (K)</td>
<td>(s) (K)</td>
</tr>
<tr>
<td>Base</td>
<td>79 400</td>
<td>182 713</td>
<td>257 1060</td>
<td>342 1090</td>
<td>$\infty$</td>
</tr>
<tr>
<td>MX</td>
<td>11 110</td>
<td>20 123</td>
<td>76 369</td>
<td>38 152</td>
<td>47 249</td>
</tr>
<tr>
<td>Ratio</td>
<td>7.2 3.6</td>
<td>9.1 5.8</td>
<td>3.4 2.9</td>
<td>9.0 7.2</td>
<td>—</td>
</tr>
</tbody>
</table>

6.1.3 Experimental Results

Table 6.1 shows the results of applying the technique to the TCAS II model. The last property was infeasible to check without pruning. For other properties, the speedup obtained was as much as a factor of 9.

6.2 Oblivious vs. Non-Oblivious Synchronization

Although the technique above works well for TCAS II, a limitation is that it is only applicable if the events are mutually exclusive. However, sometimes transitions are mutually exclusive even though their trigger events are not, as in the case of our EPD model. Before we see a technique to overcome this restriction, we first look at another performance issue, and then tackle these two problems together in Section 6.3. Specifically, we will see in this section that two models with similar intuitive behaviors but written in different styles can incur dramatically different costs in the analyses.

Figure 6.2 shows a system whose stable-state behaviors are identical to the one in Figure 6.1. The main difference between the two systems is that in Figure 6.1 event $x_1$ always occurs after $x_0$ occurs, while in Figure 6.2, event $x_1$ is not generated if there is no state change in machine $A_1$. In other words, in Figure 6.2 an event signals the completion of a state machine's execution, and the sequence of events generated is independent of what happens locally in the state machines; we say that such systems have oblivious synchronization. In Figure 6.1, an event signals a state change, and the sequence of events generated depends on which local transition is taken; we call such systems non-oblivious. Despite the differences, the behaviors of the two systems are identical as far as stable states are concerned.

A few observations are worth noting. In the non-oblivious system, the events are used for both synchronization (executing machine $A_2$ after machine $A_1$) and local control (directing machine $A_2$ to the appropriate local state), and the specifier is more concerned about the local, microstep-level interaction between the two machines. In contrast, in the oblivious system, events are merely used for synchronization; the local control logic is specified in the
guarding conditions, and the specifier foresees the overall control flow between the machines in a macrostep and constructs events to sequence the machines in the desired order. While the EPD model and virtually all of the STATEMATE machines that we have seen are not oblivious, the portion of the RSML specification of TCAS II that we analyzed (and in fact most of the entire specification) is oblivious.\textsuperscript{1} This is consistent with Harel and Naamad’s comments that in RSML a macrostep appears to be the “basic operation,” while in STATEMATE a microstep is the basic operation [73, p. 323]. Notice, however, that the differences arise not from the semantics of the language, which is often the topic of heated debate, but from the distinct mental models of the system that the specifiers have. But we note that the oblivious style of synchronization is so prevalent in TCAS II that RSML has special syntactic features to support it: Because a state change cannot be detected by a trigger event, the Prev function is used extensively to reference the previous states directly. Furthermore, so-called “identity transitions” are specified in a separate table to avoid cluttering the diagrams with self-loops as in Figure 6.2.

6.2.1 Difference in Efficiency of Analyses

Whether one style is better than the other for specification purposes is beyond the scope of this thesis. Rather, we are interested in the performance difference in model checking. We have observed in our experiments that the TCAS II model, written in the oblivious style, is more efficient to analyze than the non-oblivious EPD model. Indeed, while many properties of TCAS II could be checked in our initial study, none of the nontrivial analyses of the EPD system were feasible without using the optimizations presented later, even though the number of state variables in the EPD model is only half of that of the TCAS II model.

\textsuperscript{1}Indeed, the simple control flow prompted Leveson et al. [108] to get rid of events altogether in their new requirements language SpecTRM-RL.
Intuitively, the decoupled synchronization and local control of oblivious systems induces fewer dependencies among the state variables, potentially keeping the BDDs smaller. In particular, non-oblivious systems have many more ways to finish a macrostep than oblivious ones, and a backward search from the stable states needs to capture all these possibilities, producing larger BDDs. In addition, because each macrostep in an oblivious model has the same length, a search that is breadth-first with respect to microsteps is also breadth-first with respect to macrosteps. However, for non-oblivious models, in which the lengths of the macrosteps vary, the search is not breadth-first with respect to macrosteps, reducing the "regularity" in the state sets.

To elaborate, we extend $\prec$ (first defined in Section 6.1.2) to a binary relation over the events $E$ together with a special symbol $\bot$, which intuitively represents stable states: In addition to the definition given earlier, for each event $e$, we have $e \prec \bot$ if there exists a global state $q$ in which $e$ occurs and $q$ has a stable successor state. Figure 6.3 shows the extended precedence relations for the non-oblivious and oblivious systems in Figures 6.1 and 6.2. Note that there are fewer edges pointing to $\bot$ in Figure 6.3(b); for example, the edge $(x, \bot)$ is absent because $x$ always triggers $y$ and thus never immediately results in a stable state.

To see that this makes a difference in the analysis, let us trace what happens when we search backward from the stable states. Figure 6.4 shows the intuition for the non-oblivious model. In Figure 6.4(a), we lay down every possible sequence of events in a macrostep in the non-oblivious system; the diagram is obtained from Figure 6.3(a) by tracing all the paths starting from the external event $x_0$. The backward search can now be illustrated by traversing backward from $\bot$ in a breadth-first manner. Figures 6.4(b)–6.4(d) shows the first few iterations. In the first iteration, we visit all the stable states. In the second iteration we visit states with either $x_0$, $x_1$, or $x_2$ occurring. Because the search has now reached the beginning of the first macrostep in Figure 6.4(c), in the next iteration in Figure 6.4(d) we need to start searching backward from the end of all macrosteps again. That is, the BDD needs to represent the states in different macrosteps, possibly resulting in a loss in regularity and blowing up the BDD as a result.

Figure 6.5 shows the search for the oblivious model. Note that the sequence of events generated is always the same, so the search seems simpler. For example, in the second iteration we visit only states in which $x_2$ occurs, as opposed to states in which either $x_0$, $x_1$, or $x_2$ occurs as in the other case. (The BDD there is larger because of the additional constraints from $x_0$, $x_1$, and the state machines triggered by them.) More importantly,
(a) All event sequences

(b) Iteration 1

(c) Iteration 2

(d) Iteration 3

Figure 6.4: Backward search for the non-oblivious model

(a) Only one possible event sequence

(b) Iteration 1

(c) Iteration 2

(d) Iteration 3

Figure 6.5: Backward search for the oblivious model
Table 6.2 Performance (time in seconds) for computing fixed points for the parameterized examples

<table>
<thead>
<tr>
<th>$n$</th>
<th>Non-Oblivious &amp; Oblivious</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>base MX MC base MX MC</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>10</td>
<td>4 2 1 1 1</td>
</tr>
<tr>
<td>15</td>
<td>55 33 3 7 4</td>
</tr>
<tr>
<td>20</td>
<td>358 133 7 27 19 12</td>
</tr>
</tbody>
</table>

because every macrostep here has the same length, the search is breadth-first with respect to macrosteps as well as microsteps, making the state sets in the traversal more regular.

6.2.2 Experimental Results

As mentioned, although the analyses of the oblivious TCAS II model were generally successful, our initial attempt to analyze the non-oblivious EPD model failed miserably—even trivial properties could not be analyzed in hours of CPU time and hundreds of megabytes of memory. It is conceivable that the difference in performance is due to factors other than the synchronization styles, but we can confirm through a simple experiment that the styles can indeed have a large impact. We scaled up the systems in Figures 6.1 and 6.2 in the obvious way—we increased the number of state machines to a parameter $n$, and composed the machines in a serial fashion. We checked whether it is possible to have machine $A_{n-1}$ in state 0 and machine $A_n$ in state 1 when the system is stable. Appendix A.2 lists the SMV programs used.

Table 6.2 summarizes the results. We analyzed every model without any optimizations (base), and with pruning (MX). (The column marked MC will be explained in Section 6.3.) The results show that the non-oblivious models can be much less efficient to verify—the time required to compute the fixed points for the models with 20 state machines differs by an order of magnitude in the base case. Pruning using mutually exclusive events (MX) facilitates the analyses of every model: There is up to a factor of 2.7 reduction in time. But the gap between the two styles remains large.

Note that we cannot simply add self-transitions to turn a non-oblivious model into an oblivious one, because the extraneous events generated can potentially change the behaviors of the system. We need a more sophisticated technique to make the analyses of non-oblivious models more efficient.

6.3 Microstep Counter

In Sections 6.1 and 6.2, we saw two reasons for large BDDs. Armed with those intuitions, we attack the problems by systematically modifying the transition system to prune backward searches and to decouple the synchronization from the local control, while preserving the semantics of the model. We achieve this by incorporating a microstep counter into the system, and making every macrostep equal in length. The counter is oblivious in that its
behaviors do not depend on the internal events or the state machines, and is used to guard every local transition.

### 6.3.1 Construction of the Microstep Counter

In Section 6.1 we have defined a set $\sigma(e)$ for each event $e$ such that it contains an integer $i$ if event $e$ can occur just before the $i^{th}$ microstep. Observe that the maximum length $k$ of a macrostep is the largest integer in $\sigma(e)$ for any $e$. For Figure 6.1, we have $\sigma(x_0) = \{1\}$, $\sigma(x_1) = \{2\}$, $\sigma(x_2) = \{3\}$, and therefore $k = 3$. Note that some macrosteps may have fewer than $k$ microsteps.

Now, to symbolically encode a statecharts model as a transition system, in addition to the usual state variables defined in Section 3.3, we define a microstep counter $mc$ to range from 0 to $k$. The behavior of the microstep counter depends only on the set $\text{External}$ of external events.

**Modification 6.1** [Microstep counter] Add the following rule to the translation rules in Section 3.3:

18. Define microstep counter:

```plaintext
VAR mc: 0..k;
ASSIGN
  init(mc) := case
    no-ext-evt: 0;
    1: 1;
  esac;
  next(mc) := case
    mc = 0 & no-ext-evt: 0;
    mc = 0 & !no-ext-evt: 1;
    mc = k: 0;
    1: mc + 1;
  esac;
DEFINE no-ext-evt := $\forall e \in \text{External} e$;
```

Stability now depends only on the microstep counter:

**Modification 6.2** [Stability] Replace Rule 16 on page 25 with Rule 16':

16' DEFINE

```plaintext
stable := mc = 0;
```
The new rules intuitively say the following: If no external event occurs in the initial state, then the system is considered stable and \( mc \) is initialized to 0. Whenever some external event occurs, \( mc \) becomes 1 in the same state and a macrostep begins. The value of \( mc \) is then incremented by 1 in every subsequent microstep until the value reaches \( k \). At that point, it will be reset to 0 in the successor states, and the system will be stable. Note that the internal events and the local states do not come into the picture, and that every macrostep has exactly \( k \) microsteps.

Clearly, the local transitions in the statecharts are unaffected by the changes, but the stable states may be delayed as illustrated in Figure 6.6—when the original system is stable, the modified system may still be incrementing \( mc \). However, because the microstep counter is not visible to the user, the modified system will not produce any visible change until stable. Formally, the system \textit{stutters} in the interim \cite{105}, and all “stutter-invariant” CTL formulas, including those without the next-time \textit{X} operator, are preserved by stuttering \cite{21}. (Formulas with the \textit{X} operator can count the number of microsteps and thus may not be preserved.)

Our final modification uses the microstep counter to guard transitions.

**Modification 6.3** [Guards] Replace Rule 10 on page 25 with Rule 10′:

10′ For each \( t \in \text{Trans} \):

\[
\text{DEFINE } t := \text{in-svc}(t) \& mc \text{ in } \sigma(\text{trig}(t)) \& \text{trig}(t) \& \text{cond}(t);
\]

Note the extra conjunct \( mc \) in \( \sigma(\text{trig}(t)) \). One can intuitively think of the new rule above as changing a transition label from

\[
e_1[\text{cond}]/e_2 \quad \text{to} \quad mc \in \sigma(e_1)[\text{cond}]e_2.
\]

In other words, the trigger event \( e_1 \) becomes part of the logic of the guarding condition, and the transition is now triggered by the oblivious microstep counter. It is in this sense that we think of the modification decouples the local logic from the synchronization. Notice,
however, that this modification cannot affect the system’s behavior, because in any reachable state, the occurrence of $e_1$ implies $mc \in \sigma(e_1)$. This can be proved by induction on the definition of $\sigma$. So the inclusion of $mc \in \sigma(e_1)$ is redundant as far as forward behaviors are concerned. We make the following claim:

Claim 6.1 (Correctness) If the relation $\prec$ is acyclic, then Modifications 6.1–6.3 preserve every CTL formula that does not contain the $X$ operator and does not refer to the value of $mc$ (except in indirectly comparing it with 0 by referencing stable).

6.3.2 Benefits of the Microstep Counter

To see how these modifications help, consider again our non-oblivious system in Figure 6.2 on page 59. Figure 6.7 shows the modified machines with transitions guarded by the microstep counter. In this new system, we no longer have $x_0 \prec \bot$, because by the construction of $mc$, the external event $x_0$ can only occur when $mc = 1$, but the system is stable only when $mc = 0$ (which cannot happen immediately after $mc = 1$). Similarly, we rule out $x_1 \prec \bot$. So the relation $\prec$ now becomes exactly the same as the one in Figure 6.3(b) instead of Figure 6.3(a). Figure 6.8 shows what the event sequences look like in this new machine. Because every macrostep is identical in length, the search becomes breadth-first with respect to macrosteps (in addition to microsteps), just as in the case for the oblivious system.

$$x_0 \rightarrow \emptyset \rightarrow \emptyset \rightarrow \bot$$

$$x_0 \rightarrow x_1 \rightarrow \emptyset \rightarrow \bot$$

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \bot$$

Figure 6.8: Event sequences of the non-oblivious system with microstep counter. $\emptyset$ indicates a stuttering state in which no events occur.
Table 6.3 Performance of using microstep counter for the EPD model. Without the microstep counter, every formula could not be evaluated within two hours of CPU time. Explained in Section 5.3, the properties from left to right assert that the main buses can tolerate one failure, power sources are separated when there are no failures, all buses are powered when there are no failures, power persists unless failures occur, and that backup buses can tolerate two failures. The order refers to the relative difficulty (based on time) in analyzing them. The rightmost column will be explained in Section 6.4.

<table>
<thead>
<tr>
<th></th>
<th>(5.3)</th>
<th>(5.2)</th>
<th>(5.1)</th>
<th>(5.5)</th>
<th>(5.4)</th>
<th>forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (sec.)</td>
<td>49</td>
<td>54</td>
<td>85</td>
<td>89</td>
<td>462</td>
<td>3806</td>
</tr>
<tr>
<td>node (K)</td>
<td>464</td>
<td>402</td>
<td>837</td>
<td>452</td>
<td>2965</td>
<td>3865</td>
</tr>
</tbody>
</table>

To see that the modifications help prune unreachable simultaneous transitions in backward searches, observe that the microstep counter in Figure 6.7 makes it explicit that the transitions in machines $A_1$ and $A_2$ are mutually exclusive. The technique here is more general than using mutual exclusion of events in Section 6.1. For example, if in some system we have $\sigma(e_1) = \{1, 2\}$ and $\sigma(e_2) = \{2\}$, then the microstep counter makes it clear that transitions triggered by $e_1$ and $e_2$ cannot be enabled simultaneously when $mc = 1$, even though $e_1$ and $e_2$ may not always be mutually exclusive (i.e., when $mc = 2$).

Because our construction of the microstep counter makes certain macrosteps longer, the technique generally results in an increased number of iterations to reach a fixed point, affecting the performance in a negative way. Nevertheless, this impact is usually negligible compared with the benefits of reducing the BDD size. The lengthened macrosteps also introduce extra states in a counterexample, but these states are easy to detect and can be removed to recover the actual counterexample.

6.3.3 Experimental Results

The use of microstep counters was crucial for the analyses of the EPD model discussed in Chapter 5—none of the interesting properties could be analyzed within two hours of CPU time without using the optimization. Table 6.3 shows the results of the technique. All the searches were performed on the model without fixing the bugs B1 or B2.

The columns marked MC in Table 6.2 on page 62 show the results of applying the technique to the parameterized examples. (Appendix A.2 gives the SMV code used in the experiment.) The microstep counter dramatically improved the performance for the non-oblivious models by up to a factor of 19, making them more efficient to analyze than the oblivious models. The slight advantage of the non-oblivious models here stems from the fewer number of state variables, because the previous states of the machines are not encoded.

6.3.4 Condition-Driven Transitions

We have been assuming the absence of condition-driven transitions (Section 3.3.7) so far in this chapter. The advantage of the assumption is that we can reason about transitions by analyzing events, which are usually fewer in number than transitions. Extending our
techniques to handle condition-driven transitions requires a more general framework. The basic idea is to look at the precedence relation of transitions instead of events. Assume that we have an acyclic binary relation $\sqsubseteq$ over the transitions such that $t_1 \sqsubseteq t_2$ if $t_2$ may be taken one microstep after $t_1$ is taken. Instead of using $\sigma$, we define $\rho$ to be a mapping from the set of transitions to integers such that $\rho(e)$ is the smallest set of integers with

1. $1 \in \sigma(t)$ if $t$ has an external event as its trigger event, or $t$ does not have a trigger event but its guarding condition mentions an environmental input; and
2. for each $t$, if $i \in \sigma(t)$ then $i + 1 \in \sigma(t')$ for each $t'$ with $t \sqsubseteq t'$.

Similar to what we have before, $i$ is in $\rho(t)$ if $t$ can be taken in the $i^{th}$ microstep of some macrostep. The maximum length $k$ of a macrostep is then the largest integer in any $\rho(t)$. Now, in Modification 6.3, we simply use $\rho(t)$ instead of $\sigma(trig(t))$.

It remains to show how we obtain such a transition precedence relation $\sqsubseteq$. Recall from Section 3.3.1 on page 20 that $trig(t)$, $acts(t)$, and $cond(t)$ denote the trigger, actions, and guarding condition of the transition $t$ respectively, and $Exits(t)$ and $Enters(t)$ denote the set of local states that may be exited and entered upon taking $t$ respectively. One way to define $\sqsubseteq$ is to say $t_1 \sqsubseteq t_2$ if one of the following is true:

1. $t_2$ has a trigger event, and $trig(t_2) \in acts(t_1)$.
2. $t_2$ is condition-driven, and $src(t_2)$ is in the set $Enters(t_1)$.
3. $t_2$ is condition-driven, and one of the local states that appears in $cond(t_2)$ is in the set $Exits(t_1) \cup Enters(t_1)$.

The first condition is similar to the definition of $\prec$. If $t_2$ is condition-driven, there are two ways that $t_1$ may precede $t_2$: Either taking $t_1$ causes the system to enter the source state of $t_2$, or it affects the guarding condition of $t_2$. These are captured by the last two conditions above.

Although this definition is correct, it happens to be too conservative for our EPD model. For example, in Figure 5.2 on page 50, the definition above will assume that the two transitions in machine A precede each other, which is not possible if condition $c$ does not mention machine A. As a result, $\sqsubseteq$ is cyclic by this definition. So we strengthen the second condition above as:

2'. $t_2$ is condition-driven, $src(t_2)$ is in the set $Enters(t_1)$, and
   - either $cond(t_1)$ and $cond(t_2)$ are not mutually exclusive,
   - or $cond(t_2)$ mentions a state in $Exits(t_1) \cup Enters(t_1)$.

By this definition, the relation $\sqsubseteq$ for the EPD system is acyclic. This is the definition that we used in case study of the EPD system.
6.4 Forward vs. Backward Traversals

So far, we have been focusing on searching from the set $E$ of error states backward to find the set $I$ of initial states. Clearly, an alternative approach is to compute a fixed point forward from the initial states. More explicitly, recall that $Q$ is the set of global states, $R \subseteq Q \times Q$ is the set of global transition relations, and that

$$Pre(S) = \{ q \in Q \mid \exists q' \in S. (q, q') \in R \}.$$

In the backward approach we check whether $I \cap Pre^*(E)$ is empty. For the forward approach, we define $Succ(S)$ as

$$Succ(S) = \{ q' \in Q \mid \exists q \in S. (q, q') \in R \},$$

the set of states reachable from $S$ in one transition, and define $Succ^*(I)$ as the least fixed point of $\lambda Y. I \cup Succ(Y)$, or the set of reachable states. An error state is then reachable if and only if the intersection of $E$ and $Succ^*(I)$ is not empty.

6.4.1 Performance Difference Between Forward and Backward Traversals

Although forward and backward traversals are similar in principle, this forward approach performed poorly in our case studies—the model checker was unable to compute the reachable states within hours of CPU time. A backward traversal often takes fewer iterations to reach a fixed point than a forward traversal, because the set of error states is usually more general than the set of initial states. However, the problem here is not the number of iterations, but rather, the size of the BDDs generated. In our experiments, the BDDs generated in backward traversals usually have between hundreds to at most tens of thousands of BDD nodes, while in forward traversals, they can be two or more orders of magnitude larger. The EPD model with the microstep counter was the only model in our two case studies that was feasible to analyze using a forward search. But even so, as shown in the column “forward” in Table 6.3, it was at least an order of magnitude slower than backward traversals. Nevertheless, the verification of many hardware systems tends to benefit, rather than suffer, from forward traversals [e.g., 12, 98].

Partly inspired by Hu and Dill [91], we believe that the inefficiency is mainly due to the complicated invariants of TCAS II and the EPD system, which are maintained by forward but not backward traversals. Although, as argued in Section 6.1, invariants can sometimes improve search efficiency, paradoxically, keeping all invariants can hurt performance. Consider again machine $A_1$ in Figure 6.2. If event $x_1$ is only generated in $A_1$, then an invariant of the system is that, whenever event $x_1$ has just occurred, machine $A_1$ is in state 0 if and only if condition $c_1$ is true. If the BDD for $c_1$ is large, so will the BDD for the invariant. There are likely to be many such implicit invariants in the system, and their conjunction may have a large BDD representation even if they individually induce small BDDs. In addition, invariants may globally relate different state machines, something also likely to result in large BDDs. Forward traversals maintain all such invariants, so intuitively the BDDs for forward traversals tend to blow up in size. In particular, the set of reachable states is exactly the conjunction of all invariants of the system, so its BDD is likely to be large too,
1. Let $Q_0$ be any nonempty subset of $Pre^*(E) \cap I$. Iteratively compute $Q_{i+1} = Suc(Q_i) \cup Q_i$ until reaching $E$.

2. Start with some $q_m \in Q_m \cap E$ and iteratively pick some $q_{i-1} \in Pre(q_i) \cap Q_{i-1}$ to obtain a counterexample $q_0, q_1, \ldots, q_m$.

Figure 6.9: Original Algorithm for counterexample search

making the computation of reachable states an intractable operation. In low-level hardware verification, the BDDs often remain small, because each invariant is usually localized and involves only a small number of state variables. This is not the case in our statecharts models however.

For backward traversals, the situation is quite different. For example, there are no counterparts of the invariant mentioned above when backward traversals are used, because the truth value of $c_1$ does not determine the state of the system before the microstep. Certainly, some different (backward) “invariants” are maintained in backward traversals, but they tend to depend on the states from which the search starts, and for our systems their BDDs tend to be smaller (or can be made smaller using the techniques presented).

6.4.2 Implications on Counterexample Search

In addition to fixed-point computations, this performance difference in forward and backward traversals also has an impact on counterexample search—during our initial analysis of TCAS II, we found that when a property was disproved in a few minutes using backward search, finding a counterexample might take hours. The reason is that SMV uses a forward search to find counterexamples, and suffers from the BDD blowup pointed out above.

Figure 6.9 shows the original counterexample search algorithm used in the model checker. The set $Q_0$ can be any nonempty subset of the intersection, but it is convenient to choose $Q_0$ to be an arbitrary singleton set. The set $Q_i$ is the states that are reachable from $Q_0$ in at most $i$ transitions. We obtain a counterexample by tracing backward from $Q_m \cap E$. 
Start with some \( q_0 \in Y_n \cap I \) and iteratively pick some \( q_i \in \text{Suc}(q_{i-1}) \cap Y_{n-i} \) to obtain a counterexample \( q_0, q_1, \ldots, q_n \).

![Diagram](image)

Figure 6.10: Modified algorithm for counterexample search. The sets \( Y_i \) are computed in Figure 2.1 on page 8.

The first, forward traversal in Figure 6.9 was the bottleneck. The sequence of successor state sets required large BDDs. To solve the problem, our colleague Steve Burns modified the counterexample search routine in the model checker, resulting in substantial speedup. The idea, illustrated in Figure 6.10, is to remember every \( Y_i \) computed in Figure 2.1 on page 8 (our actual implementation stores the difference \( Y_i \setminus Y_{i-1} \) instead of \( Y_i \)), and use them to restrict the counterexample search. This same algorithm is used in other model checkers as well [e.g., 91]. Note that although a forward traversal is still required, the BDDs produced are much smaller, because at each step only a single state is involved in the computation of successors. Before, the counterexample search would spend hours of CPU time and hundreds of megabytes of memory, and still could not finish. Now a counterexample can be found in just a few seconds.

6.5 Discussion and Related Work

The techniques described in this chapter aim at reducing the size of the BDDs representing state sets. In hardware verification, techniques with the same goal exist and usually work by altering these BDDs during the search. Some of these techniques try to exploit special structures in the circuits, such as symmetries [42] or asynchrony [2]. Because our models lack these special structures, these techniques are not applicable. Other hardware techniques work for general classes of circuits [e.g., 32, 91, 126]. We applied some of them to our models, but the results were not satisfactory. However, we note that our method of pruning backward traversals using invariants is similar in spirit to the work on hardware verification by Cabodi et al. [31], who propose doing an approximate forward traversal reachable states, which is then used to prune backward traversals. (An invariant is a superset of the reachable states.) Their method is more general, but more expensive because an extra symbolic traversal is required. Yang et al. [138] also try to exploit invariants of a certain form, but they report that their methods are not effective to our TCAS II model.

In general, the techniques we developed concentrate on statecharts. Intuitively, our two techniques use information from forward analysis on event precedence to prune backward searches, and to realign the search frontiers. This strategy of combining forward syntactic
analysis and backward searches appears to be a promising approach to improving the efficiency of symbolic model checking. The method of microstep counter also differs from the other approaches above in that it changes the underlying transition system before applying model checking.

Alur et al. [6] give an algorithm for model checking in the presence of transition hierarchies, for which the distinction between microsteps and macrosteps is a special case. Their algorithm uses a nested forward search to explore all the microsteps in a single macrostep before proceeding to the next macrostep. They also observe that if we are only concerned about stable-state behaviors, the states visited in the inner loop can be safely discarded to conserve memory, at the expense of additional computation time. We experimented with the idea, but found the tradeoff bad for our models.

Getting intuition on BDD size in general is notoriously hard, because the size does not directly correlate to simple measures such as the number of variables or reachable states. However, formal software specifications are often written in a few common styles or using a few popular idioms, and it may be possible to gain enough insights to optimize for these common cases. This work follows this direction and contributes to a better understanding of how various ways of specification affects the efficiency of verification. We hope that the results will not only be useful for developing more efficient model-checking algorithms, but also be valuable for designing specifications or specification languages that are more amenable to symbolic model checking.
Other Optimizations

Apart from reducing the BDD size for state sets, we can improve the performance of symbolic model checking by reducing the BDD size for the transition relation, by removing system components that are irrelevant to the property being checked, or by reducing the number of search iterations. We will look at each of these techniques in this chapter. Preliminary results appeared in Chan et al. [37].

7.1 Partitioning Transition Relations

A common bottleneck of model checking is the BDD size for the transition relation, which can be reduced by conjunctive or disjunctive partitioning [27]. The former can be used naturally for statecharts, and we have modified SMV to partition the transition relation more effectively. We also apply disjunctive partitioning, which is normally used only for asynchronous systems. Combining the two techniques, we obtain DNF partitioning. As we will see, the issues in this section are not only the BDD size for the transition relation, but also the size of the intermediate BDDs generated for each predecessor computation. We will also show some experimental results of partitioning the TCAS II model in various ways.

7.1.1 Background

We first review the idea of conjunctive and disjunctive partitioning. The transition relation $R$ is sometimes given as a disjunction $D_1 \lor D_2 \lor \cdots \lor D_j$, and the BDD for $R$ can be huge even though each disjunct has a small BDD. So instead of computing a monolithic BDD for $R$, we can keep the disjuncts separate. The predecessor computations can be easily modified by distributing the existential quantification over the disjunction. We thus have

$$
Pre(S) = \exists X'. R(X, X') \land S(X')
$$

$$
= \exists X'. (D_1(X, X') \lor D_2(X, X') \lor \cdots \lor D_j(X, X')) \land S(X')
$$

$$
= d_1(X) \lor d_2(X) \lor \cdots \lor d_j(X)
$$

where for $1 \leq i \leq j$,

$$
d_i(X) = \exists X'. D_i(X, X') \land S(X').
$$
So we can compute the predecessors without ever building the BDD for $R$. Successor computation is symmetric. 

If $R$ is given as a conjunction $C_1 \land C_2 \land \cdots \land C_k$, we can still keep the conjuncts separate as above, but predecessor computations become more complicated. The problem is that existential quantification does not distribute over conjunctions, so it appears that we have to compute the BDD for $R$ anyway before we can quantify out the variables. A trick to avoid this is early quantification. Define $X'_1, X'_2, \ldots, X'_k$ to be disjoint subsets of $X'$ such that their union is $X'$ and for $1 \leq i \leq k$, the conjunct $C_i$ does not depend on any variable in $X_p$ for any $p < i$. We compute

$$c_1(X, X') = \exists X'_1. C_1(X, X') \land S(X')$$
$$c_2(X, X') = \exists X'_2. C_2(X, X') \land c_1(X, X')$$
$$\vdots$$
$$Pre(S) = c_k(X) = \exists X'_n. C_k(X, X') \land c_{k-1}(X, X').$$

The intuition is to quantify out variables as early as possible, and hope that each intermediate $c_i$ for $1 \leq i < k$ remains small. The effectiveness of the procedure depends critically on the choice and ordering of the conjuncts $C_1, C_2, \ldots, C_k$.

### 7.1.2 Determining a Conjunctive Partition

We cannot easily construct the monolithic BDDs for the transition relations for our TCAS II and EPD models, but each transition relation is naturally specified as a conjunction, so we can use conjunctive partitioning. Although SMV supports this feature, it determines the partition in a simplistic way: Recall from Section 2.3 that an SMV program consists of a list of parallel assignments, whose conjunction forms the transition relation. The model checker constructs the BDDs for all assignments, and incrementally builds their conjunction in the (reverse) order they appear in the program. In this process, whenever the BDD size exceeds a user-specified threshold, it creates a new conjunct in the partition. So the partition is solely determined by the syntax, and no heuristic or semantic information is used.

To better determine the partition, we changed the model checker to allow the user to specify the partition manually. We also implemented in the model checker a variant of the heuristics by Geist and Beer [63] and Ranjan et al. [125] to automatically determine the partition. The central idea behind the heuristics is to greedily select conjuncts that allow early quantification of more variables while introducing fewer variables that cannot be quantified out. Our implementation of the heuristics worked quite well. The partitions generated compared favorably with, and sometimes outperformed, the manual partitions that we tried.

### 7.1.3 Disjunctive Partitioning

Disjunctive partitioning is superior to conjunctive partitioning in the sense that ordering the disjuncts is less critical, and that each intermediate BDD is a function of $X$ (instead of $X \cup X'$) and thus tends to be smaller.
Unfortunately, when the transition relation $R$ is a conjunction, in general there are no simple methods for converting it to a small set of small disjuncts. If we define a cover $\alpha_1(X,X'), \alpha_2(X,X'), \ldots, \alpha_j(X,X')$ whose disjunction is a tautology, then we can indeed disjunctively partition $R$ by distributing $R$ over the cover:

$$R = (\alpha_1 \lor \alpha_2 \lor \cdots \lor \alpha_j) \land R$$

$$= D_1 \lor D_2 \lor \cdots \lor D_j$$

where for $1 \leq i \leq j$,

$$D_i = \alpha_i \land R = \alpha_i \land C_1 \land C_2 \land \cdots \land C_k.$$  

But for most choices of covers, each $D_i$ is still large.

For statecharts in which most events are mutually exclusive, such as the portion of TCAS II that we looked at, we can use these events, say $u_1, u_2, \ldots, u_{j-1}$, to form a cover.

$$\alpha_i = u_i \land \bigwedge_{1 \leq p < j, p \neq i} \neg u_p$$

for $1 \leq i < j$, and

$$\alpha_j = \neg u_1 \land \neg u_2 \land \cdots \land \neg u_{j-1}$$

$$\alpha_{j+1} = \neg \alpha_1 \land \neg \alpha_2 \land \cdots \land \neg \alpha_j.$$  

In other words, $\alpha_i$ corresponds to the states in which only $u_i$ has just occurred, $\alpha_j$, none of the events have, and $\alpha_{j+1}$, at least two of the events have. They clearly form a cover. We made two observations. First, we can drop $\alpha_{j+1}$, which is a contradiction because of the mutual exclusion assumption. Second, most of the parallel assignments in our SMV program are guarded by conditions on the events; for example, an assignment that models a state transition requires the occurrence of the trigger event. If the event is, say, $u_i$ for some $1 \leq i < j$, then the BDD for the assignment is applicable only to the disjunct $D_i$, and all the other disjuncts of the transition relation are irrelevant. So, each disjunct may remain small. Notice that to apply this technique, we have to find a set of provably mutually exclusive events, which can be done as described in Section 6.1.2.

7.1.4 DNF Partitioning and Serialization

A disadvantage of partitioning $R$ based on events is that the sizes of the disjuncts are often skewed. In particular, if a single event may trigger a number of complex transitions, its corresponding disjunct could be large. Figure 7.1 shows an example in which an event $x$ triggers two state machines. If all the guarding conditions are complex, the BDD for the disjunct corresponding to $x$ may be large.

One solution to this problem is to apply conjunctive partitioning to large disjuncts, resulting in what we call DNF partitioning. It uses both BDD size (as in conjunctive partitioning) and structural information (as in disjunctive partitioning) to partition the transition relation, and may perform better than relying on either alone.
Figure 7.1: One event triggering two state machines

Figure 7.2: The serialized machine
Alternatively, we may serialize the complicated microstep into cascading microsteps to reduce the BDD size. Figure 7.2 illustrates this idea. We have “inserted” a new event $u$ after $x$. Note that the resulting machine has more microsteps in a step. So although this method is effective in reducing the BDD size in this case, it often increases the number of iterations to reach a fixed point. Also, the transformation may not preserve the behavior of the system and the property analyzed. A sufficient condition is that the guarding conditions in machine $B$ do not refer to machine $A$’s local states, $x$ is mutually exclusive with all other events, and we are checking a stutter-invariant property that does not explicitly mention any of the state machines, transitions or events involved in the transformation.

7.1.5 Experimental Results

**TCAS II** Table 7.1 summarizes the results of applying the various partitioning techniques to our models of TCAS II. It shows the resources (time in seconds and number of BDD nodes used in thousands) for building the BDDs for the transition relation $R$ as well as the resources for evaluating the properties. Three models were examined. Our starting point is called the *full model*. The *mistranslated model* contains a real translation bug, and is included to give an example of analyzing a highly flawed design. The *serialized model* was obtained from the full model with one of the microsteps serialized. For each model, we performed model checking using various partitioning methods: heuristic conjunctive partitioning (CP), disjunctive partitioning (DP), and DNF partitioning (CP and DP). Recall that the last two methods can only be used with the mutual exclusion of events (MX).

Rows 1 and 3 are taken from Table 6.1 and show the results for the base case and the results with pruning using mutually exclusive events. In both cases, the conjunctive partitioning as implemented in SMV was used. Row 1 shows that the fixed-point computations for one of the properties could not be completed for the full model when we used only the conjunctive partitioning as implemented in SMV. As shown in Row 2, the savings resulting from the heuristic for conjunctive partitioning were quite significant.

Disjunctive partitioning, which must be combined with the mutual exclusion of events, appeared to be inefficient (Row 5) when compared with applying the mutual exclusion alone (Row 3). The reason is that one of the disjuncts of the transition relation was large, with over $10^5$ BDD nodes, at least an order of magnitude larger than other disjuncts; this is reflected in the table by the large number of BDD nodes needed to construct the transition relation. We conjunctively partitioned the large disjunct, leading to the more efficient DNF partitioning (Row 6). It performed marginally better than conjunctive partitioning with mutual exclusion of events (Row 4), but the space requirements were consistently lower.

To further illustrate the differences among the various partitioning techniques, we looked at a version of the model that contains a translation error from the RSML machines to the SMV program. We made this bug early in the previous study, although we soon discovered

---

1 Actually, we implemented a simple improvement that was used in all results including this base analysis. As explained in Section 2.2, predecessor computation involves a conjunction and an existential quantification. The two operations can be carried out simultaneously to avoid building the usually large conjunction explicitly [27]. SMV performs this optimization except when conjunctive partitioning is used. We simply changed SMV to eliminate this limitation.
Table 7.1 Performance of different ways of partitioning TCAS II. For each group of experiments, the best time and space requirements for each property are shown in bold face. An entry with \(\infty\) indicates timeout after one hour.

<table>
<thead>
<tr>
<th>Building BDDs for (R)</th>
<th>(4.6)</th>
<th>(4.4)</th>
<th>(4.1)</th>
<th>(4.7)</th>
<th>(4.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full Model</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of fixpoint iterations</td>
<td>24</td>
<td>29</td>
<td>29</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>Optimizations</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
</tr>
<tr>
<td>MX</td>
<td>CP</td>
<td>DP</td>
<td>(s)</td>
<td>(K)</td>
<td>(s)</td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>20</td>
<td>93</td>
</tr>
<tr>
<td>2</td>
<td>(\sqrt{\ })</td>
<td>—</td>
<td>33</td>
<td>176</td>
<td>40</td>
</tr>
<tr>
<td><strong>Mistranslated Model</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of fixpoint iterations</td>
<td>24</td>
<td>29</td>
<td>29</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>Optimizations</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
</tr>
<tr>
<td>MX</td>
<td>CP</td>
<td>DP</td>
<td>(s)</td>
<td>(K)</td>
<td>(s)</td>
</tr>
<tr>
<td>7</td>
<td>(\sqrt{\ })</td>
<td>—</td>
<td>20</td>
<td>93</td>
<td>285</td>
</tr>
<tr>
<td>8</td>
<td>(\sqrt{\ })</td>
<td>(\sqrt{\ })</td>
<td>26</td>
<td>174</td>
<td>323</td>
</tr>
<tr>
<td>9</td>
<td>(\sqrt{\ })</td>
<td>—</td>
<td>36</td>
<td>462</td>
<td>972</td>
</tr>
<tr>
<td>10</td>
<td>(\sqrt{\ })</td>
<td>(\sqrt{\ })</td>
<td>42</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td><strong>Serialized Model</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of fixpoint iterations</td>
<td>36</td>
<td>41</td>
<td>45</td>
<td>54</td>
<td>38</td>
</tr>
<tr>
<td>Optimizations</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
<td>time node</td>
</tr>
<tr>
<td>MX</td>
<td>CP</td>
<td>DP</td>
<td>(s)</td>
<td>(K)</td>
<td>(s)</td>
</tr>
<tr>
<td>11</td>
<td>(\sqrt{\ })</td>
<td>—</td>
<td>27</td>
<td>103</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>(\sqrt{\ })</td>
<td>(\sqrt{\ })</td>
<td>31</td>
<td>167</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>(\sqrt{\ })</td>
<td>—</td>
<td>27</td>
<td>139</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>(\sqrt{\ })</td>
<td>(\sqrt{\ })</td>
<td>48</td>
<td>136</td>
<td>11</td>
</tr>
</tbody>
</table>

MX: mutually exclusive events  CP: heuristic conjunctive partitioning  DP: disjunctive partitioning
Other Optimizations

it by inspection. The mistake was omitting some self-loops similar to those in Figure 6.2 on page 59, and effectively made the system non-oblivious, producing an order of magnitude increase in analysis time.

Interestingly, the particular partition generated by the the heuristic performed poorly for this model (Row 8). DNF partitioning, on the other hand, continued to give significant time and space reductions (Row 10). The miserable results of disjunctive partitioning (Row 9) were again due to the disproportionately large BDD in the partition.

We serialized a microstep in the full model to break the large disjunct into four BDDs of sizes about a hundred times smaller. Disjunctive partitioning now used less space (Rows 5 vs. 13). However, since the number of microsteps in a step increased, all checks suffered from the larger number of iterations needed to reach fixed points. They all ended up performing about the same, with disjunctive and DNF partitioning having the slight edge, particularly in the space requirements for the more difficult searches.

The data suggest that if the disjuncts are small to start with, disjunctive partitioning is a viable option, but serializing the microstep in order to use disjunctive partitioning is not advantageous in our case. In general, we find the effects of lengthening and shortening macrosteps difficult to predict. They represent a tradeoff between the complexity of predecessor computations and the number of search iterations.

EPD While the conjunctive partitioning heuristic produced some improvements for TCAS II, it was vital for our EPD analyses: Without the heuristic, the properties could not be checked in two hours of CPU time (even with microstep counters). The results shown in Table 6.3 on page 66 were obtained using the heuristic. Note also that our method of disjunctive partitioning could not be used on the EPD model, because most events there are not mutually exclusive.

7.2 Automatic Abstraction

In this section, we give a simple algorithm to remove a part of the model that cannot affect the property being checked. For example, a system may have a number of outputs (which may be local states or events). If we are analyzing only one of them, the logic that produces other outputs may be abstracted away, provided these outputs are not fed back to the system. This technique is sometimes known as cone-of-influence reduction in the literature.

7.2.1 Dependency Analysis

We determine the abstraction by a simple dependency analysis on the statecharts description. Initially, only the local states, events, transitions, or inputs that are explicitly mentioned in the property are considered relevant to the analysis. Then the following rules are applied recursively:

- If an event is relevant, then so are all the transitions that may generate the event.
- If a transition is relevant, then so are its trigger event, its source local state, and everything that appears in its guarding condition.
• If a local state is relevant, then so are all the transitions out of or into it.\textsuperscript{2}

(Note that the relevance of an input does not make any other entity relevant.) These rules are repeated until a fixed point is reached. Essentially, this is a search in the dependency graph, and the time complexity is linear in the size of the graph.

Note that the abstraction may shorten the length of a macrostep, because the machines abstracted away might still be running when the abstract machines terminate the macrostep. Indeed, in the extreme case, if the machines removed contain an infinite loop, the abstraction will remove this error and produce unsound results. However, it is easy to see that if every macrostep terminates (which can be guaranteed by the acyclicity of the event precedence relation defined in Section 6.1.2), then the abstraction preserves every stutter-invariant property. For the rest of this section, we will assume that every macrostep terminates and the property being checked is stutter-invariant.

Similar dependency analyses could also be performed by model checkers on the underlying transition system representing the statecharts, but this may not be effective. For example, an input would appear to depend on every event (Rule 15 and Rule 16 on page 25). Carrying out dependency analysis on the high-level statecharts description does not fall prey to this particular false dependency.

Other forms of false dependencies are possible, however. Suppose we are given the system in Figure 7.2 from the previous section. From the syntax, the event \( u \) appears to depend on both conditions \( a \) and \( a' \), but in fact it does not, because regardless of the truth values \( a \) and \( a' \), event \( u \) will be generated as a result of event \( x \). To detect such false dependencies, one can check whether the disjunction of the guarding conditions of the transitions out of a local state with the same trigger and action events is a tautology. This can sometimes be checked efficiently using BDDs \[77\]. However, the syntax sometimes allows easy detection of most false dependencies of this kind—for example, the self-loops in Figure 7.2 are specified in RSML as identity transitions, which can be used to infer that the occurrence of event \( u \) does not depend on conditions \( a \) and \( a' \).

Some false dependencies are harder to detect automatically. For example, the guarding conditions involved may not form a tautology, but in all reachable states, one of the guarding conditions holds whenever the trigger event occurs. As another example, in Figure 7.3, the event \( y \) does not depend on any of the guarding conditions, because it is always generated one or two microsteps after \( w \). In practice, the synchronization of the system should be evident to the designer, who may specify the suspected false dependencies in temporal logic formulas, which can be verified using model checking. If the results indeed show no real dependencies, this information can be used in the dependency analysis to obtain a smaller abstract model of the system. In our TCAS II analysis, the synchronization is simple enough that the kind of false dependencies mentioned above can be easily detected.

\[7.2.2\] **Experimental Results**

Table 7.2 shows the performance of analyzing the abstract models. The reductions obtained for TCAS II were significant (although one of the properties still could not be analyzed

\[\textsuperscript{2}\text{In the presence of hierarchical states, the superstate also becomes relevant.}\]
Figure 7.3: False dependency: Event $y$ does not depend on any guarding condition.

<table>
<thead>
<tr>
<th></th>
<th>time node bits</th>
<th>time node bits</th>
<th>time node bits</th>
<th>time node bits</th>
<th>time node bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(s) (K)</td>
<td>(s) (K)</td>
<td>(s) (K)</td>
<td>(s) (K)</td>
<td>(s) (K)</td>
</tr>
<tr>
<td><strong>TCAS II</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full</td>
<td>79 400 227</td>
<td>182 713 227</td>
<td>257 1060 227</td>
<td>342 1090 227</td>
<td>$\infty$ 227</td>
</tr>
<tr>
<td>Abstract</td>
<td>5 65 142</td>
<td>17 93 142</td>
<td>72 362 150</td>
<td>26 115 142</td>
<td>$\infty$ 150</td>
</tr>
<tr>
<td>Ratio</td>
<td>15.8 6.2 1.6</td>
<td>10.7 7.7 1.6</td>
<td>3.6 2.9 1.5</td>
<td>13.2 9.5 1.6</td>
<td>$-$ 1.5</td>
</tr>
<tr>
<td></td>
<td>(4.6)</td>
<td>(4.4)</td>
<td>(4.1)</td>
<td>(4.7)</td>
<td>(4.9)</td>
</tr>
<tr>
<td><strong>EPD</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full</td>
<td>49 464 117</td>
<td>54 402 117</td>
<td>85 837 117</td>
<td>89 452 117</td>
<td>462 2965 117</td>
</tr>
<tr>
<td>Abstract</td>
<td>36 460 70</td>
<td>40 395 70</td>
<td>85 837 117</td>
<td>66 415 70</td>
<td>462 2965 117</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.4 1.0 1.7</td>
<td>1.4 1.0 1.7</td>
<td>1.0 1.0 1.0</td>
<td>1.3 1.1 1.7</td>
<td>1.0 1.0 1.0</td>
</tr>
<tr>
<td></td>
<td>(5.3)</td>
<td>(5.2)</td>
<td>(5.1)</td>
<td>(5.5)</td>
<td>(5.4)</td>
</tr>
</tbody>
</table>
without using other optimizations). Recall that in our models, we omitted most of the
details in Other-Aircraft. Many of the outputs of Own-Aircraft that were inputs to Other-
Aircraft thus became irrelevant, unless we explicitly mentioned them in the property.

However, the reduction achieved for the EPD model was more modest because of the
higher interdependencies among the components. Three of the properties are concerned
with about the main power system, so the backup system can be removed safely. However,
no abstraction was possible for the other two properties, which depend on the whole model.

7.3 Short-Circuiting

It is easy to see that to falsify a property we do not always need to compute a fixed point.
For example, to check whether an error state is reachable using a backward search, we can
stop once an initial state is encountered. More generally, this short-circuiting technique
(sometimes also known as on-the-fly or local model checking in the literature) can be ap-
p lied to the outermost fixed point, and occasionally the inner ones. The technique may
substantially reduce the time and space used when a short counterexample exists. Table 7.3
shows the reductions obtained in our case studies.

7.4 Discussion and Related Work

Cabodi et al. [30] propose disjunctive partitioning for synchronous circuits. They require
the designer to come up with a partition manually, while we focus on statecharts and exploit
mutually exclusive events.

Using abstraction to facilitate analysis is a very old idea [44, 50], and has also been
applied to state-based software specifications. For example, Heimdahl and Whalen [78]

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 & time node & time node & time node & time node & time node \\
 & (s) (K) & (s) (K) & (s) (K) & (s) (K) & (s) (K) \\
\hline
TCAS II & (4.6) & (4.4) & (4.1) & (4.7) & (4.9) \\
\hline
Full & 79 400 & 24 & 182 713 & 29 & 257 1060 & 29 & 136 2090 & 28 & 136 751 & 24 & \infty & \infty & 26 \\
SC & 62 400 & 15 & 143 713 & 15 & 61 669 & 11 & 136 751 & 24 & \infty & \infty & 17 \\
Ratio & 1.3 1.0 & 1.6 & 1.3 1.0 & 1.9 & 4.2 1.6 & 2.6 & 2.5 1.5 & 1.6 & - & - & 1.5 \\
\hline
EPD & (5.3) & (5.2) & (5.1) & (5.5) & (5.4) \\
\hline
Full & 49 464 & 30 & 54 402 & 40 & 85 837 & 40 & 89 452 & 61 & 462 2965 & 40 \\
SC & 12 132 & 10 & 30 400 & 20 & 37 696 & 20 & 30 94 & 41 & 385 2965 & 20 \\
Ratio & 4.1 3.5 & 3.0 & 1.8 1.0 & 2.0 & 2.3 1.2 & 2.0 & 3.0 4.8 & 1.5 & 1.2 1.0 & 2.0 \\
\hline
\end{tabular}
\caption{Performance of short-circuiting. Columns marked “iter” indicate the numbers of
iterations required. SC indicates results with short-circuiting. Formula (5.5) requires two
nested fixed points to evaluate, and only the outer one was short-circuit ed. The number
of iterations reported is the sum of the two numbers. Microstep counters were used in the
EPD model, but not in the TCAS II model.}
\end{table}
use a dependency analysis technique similar to the one described Section 7.2.1, but their motivation is to facilitate manual review of the TCASI requirements, rather than automatic verification. In the context of verifying SCR requirements, Bharadwaj and Heitmeyer [14] suggest two automatic abstraction techniques, one of which is similar to our dependency analysis.

We note that in our EPD analyses, fixing the bug B1 in the model with microstep counter dramatically reduces the time taken to evaluate each formula to less than fifteen seconds. This confirms the general wisdom that design errors often introduce “irregular” behaviors to the system, resulting in large BDDs. The observation suggests early use of model checking to discover bugs as soon as possible to reduce the costs of analyzing the larger and more mature model.
Chapter 8

Nonlinear Arithmetic

Thanks to the optimization techniques in the previous two chapters, all the properties that we were interested in could be checked efficiently. We would like to expand our case studies to cover the components that were manually abstracted away, particularly the Other-Aircraft state machine in TCAS II (Section 4.2.1), but unfortunately that part contains nonlinear arithmetic constraints, which provably require exponentially large BDDs. In this chapter, we will see one approach to overcoming this problem. We focus on a restricted class of systems, which has a natural correspondence to certain statecharts and RSML models, and show how the BDD-blowup can be avoided by combining BDDs and an auxiliary constraint solver. The approach has been validated on a small example inspired by TCAS II. Preliminary results appeared in Chan et al. [36].

8.1 Overview

TCAS II falls into a large class of embedded reactive systems consisting of a finite-state control component together with numeric data inputs that measure quantities such as velocities and temperatures. The domains of these data values may be bounded or unbounded, discrete or continuous. In these systems, state transitions depend on predicates, or constraints, on these numerical values. Our approach to handling constraints in Chapter 4 exploited the finiteness of the data, representing each bit of the inputs as a BDD variable and constraints as BDDs in these variables. This worked well when dealing with purely linear constraints but did not extend efficiently to nonlinear constraints, such as those with multiplications over integer variables [23, 109].

There are at least three reasons why in general symbolic model checking for systems with nonlinear constraints is hard. The foremost problem is the lack of a succinct symbolic representation for state sets and relations. As mentioned, BDDs fail to give a compact representation when the Boolean variables represent the bits. A natural alternative is to use algebraic representations, such as DNF formulas over inequalities. However, partly because they have proved inefficient for representing control structures, it is unclear whether DNF formulas scale well for data constraints. (But we note that this approach has been used for linear constraints over reals [4] and over unbounded integers [26].)

The second and third issues lie in the manipulations of state sets. Recall that to do symbolic model checking, apart from simple Boolean operations, we need to (1) detect fixed points and (2) compute predecessors and successors. The former is equivalent to
determining the emptiness of a set. For Boolean encoding, this amounts to satisfiability checking. For arithmetic constraints, this corresponds to determining whether some Boolean combination of constraints is feasible, which in general is a harder problem. Furthermore, the computations of predecessors and successors are especially intractable. If the set $X$ of state variables contains $n$ numeric variables, then the step of successor computation

$$\exists X. R(X, X') \land S(X)$$

will involve projecting some $2^n$-dimensional set onto the $n$-dimensional space. This is in general a very expensive operation for nonlinear constraints. (For polynomial constraints over the reals, see, for example, Basu [11]. For unbounded integers, the function is even not computable.)

Our approach to attacking these problems is to focus on a class of systems with restrictions on the updates to the numeric variables: A transition must either set all new data values based only on absolute properties of their current values, or else leave them unchanged. Although even a simple update like $v_1 := v_1 + 1$ is ruled out, we do allow complicated guarding conditions such as $v_1 v_2 > v_3$ and $v_1 < \sin v_2$, and we do not limit the data domain. The restriction is satisfied by the RSML specifications that do not use the Prev construct, and by the STATEMATE models that do not have internal numeric variables. Under this restriction, we show that the data portion of the state sets arisen in the symbolic traversals can be represented as some Boolean combinations of the constraints that appear in the guarding conditions. As a result, it suffices to encode the data space as BDDs in which each Boolean variable represents one of these constraints (as opposed to a bit of an integer variable as we did originally). This solves the first problem pointed out above.

The most noted property of our class of systems is that, to compute predecessors or successors, we do not need to perform projection over the numeric variables as suggested above. Instead, we can reduce the problem to performing projection over Boolean functions (as in conventional BDD-based model checking) and determining the feasibility of conjunctions of nonlinear constraints. To determine feasibility, we rely on existing algorithms from the literature of nonlinear constrained optimization and nonlinear constraint logic programming. A simple approach to combining such a constraint solver and the model checker is to test all combinations of constraints for feasibility before applying model checking. We develop a potentially more efficient approach whereby we prune the infeasible paths from the BDDs on the fly. We will describe both approaches in this chapter.

8.2 System Models

We first give the usual definitions of labeled transition systems, bisimulation equivalence, and quotient systems from the literature. Then we define our model of constrained transition systems, whose semantics can be defined in terms of labeled transition systems, and then show that there is a natural finite bisimulation for every constrained transition system.
8.2.1 Labeled Transition Systems

We extend our definition of transition systems to include labels on states. Formally, a labeled transition system is a tuple \((Q, R, I, \Sigma, L)\), where \(Q\), \(R\), and \(I\) are as defined in Section 2.1, \(\Sigma\) is a set of atomic propositions, and \(L: Q \rightarrow \mathcal{P}(\Sigma)\) labels each state with the set of atomic propositions in \(\Sigma\) that are true at that state. When a CTL formula is evaluated in a labeled transition system, we require that the atomic propositions in the formula are drawn from \(\Sigma\).

Intuitively, an observer sees the label of the current state, but not the state itself. Two states are indistinguishable if their labels are the same and their successors are again indistinguishable. Formally, we say that an equivalence relation \(\approx\) of \(Q\) is a bisimulation [cf. 113, p. 42] if for all states \(q_1\) and \(q_2\), we have that \(q_1 \approx q_2\) implies all of the following:

1. \(L(q_1) = L(q_2)\).
2. \(q_1 \in I\) iff \(q_2 \in I\).
3. For all \(q'_1 \in Q\) with \((q_1, q'_1) \in R\), there exists a \(q'_2 \in Q\) with \((q_2, q'_2) \in R\) and \(q'_1 \approx q'_2\).
4. For all \(q'_2 \in Q\) with \((q_2, q'_2) \in R\), there exists a \(q'_1 \in Q\) with \((q_1, q'_1) \in R\) and \(q'_1 \approx q'_2\).

The quotient system of \((Q, R, I, \Sigma, L)\) with respect to a bisimulation \(\approx\) is a labeled transition system \((Q^{\approx}, R^{\approx}, I^{\approx}, \Sigma, L^{\approx})\). The quotient state space \(Q^{\approx}\) is the set of equivalence classes induced by \(\approx\). For all \(S\) and \(S'\) in \(Q^{\approx}\), we have \((S, S') \in R^{\approx}\) if and only if there exist \(q \in S\) and \(q' \in S'\) with \((q, q') \in R\). We define \(L^{\approx}(S) = L(q)\) for any \(q\) in \(S\), and \(I^{\approx} = \{S \in Q^{\approx} \mid S \cap I \neq \emptyset\}\). We say that \(\approx\) is finite if \(Q^{\approx}\) is finite.

The correctness of our approach relies on the following well-known result [see, e.g., 21].

**Lemma 8.1** Any CTL formula holds in a labeled transition system \(M\) if and only if it holds in the quotient system of \(M\) with respect to any bisimulation.

8.2.2 Constrained Transition Systems

We are interested in reactive systems with a finite control component and a finite or infinite data component. Formally, a constrained transition system is a tuple \((N, N_0, V, D, \Delta, C)\), where

- \(N\) is a finite set of control nodes.
- \(N_0 \subseteq N\) is a set of initial control nodes.
- \(V\) is a finite set of data variables. The domain of each variable is unrestricted.
- \(D\) is the cross product of the domains of the data variables.
- \(C\) is a finite set of atomic constraints on \(V\), where an atomic constraint on \(V\) is a decidable predicate over the data variables in \(V\).
- \(\Delta\) is a mapping from \(N \times N\) to constraints over \(V \cup V'\), where
  - \(V'\) is the primed version of \(V\), and as in Section 2.2, represents the next-state versions of the current-state data variables in \(V\).
A constraint is any finite Boolean combination of atomic constraints.

We require each $\Delta(n, n')$ to have a special form described below.

Note that a valuation of $V$ corresponds to an element in $D$, and a constraint on $V$ corresponds to a subset of $D$. If $c(V)$ is a constraint on $V$, we denote by $\|c(V)\|$ the set of elements in $D$ that satisfy $c(V)$. The constraint $c(V)$ is feasible if $\|c(V)\|$ is not empty. We write $c$ for $c(V)$ when there is no ambiguity. Note that for any $n, n' \in N$, the set $\|\Delta(n, n')\|$ is a subset of $D \times D$.

The constrained transition system defines a labeled transition system $(Q, R, I, \Sigma, L)$ as follows.

- The state space $Q$ is $N \times D$.
- The set $I$ of initial states is $N_0 \times D$.
- The transition relation $R$ is defined such that for all $(n, a)$ and $(n', a')$ in $N \times D$, the state $(n', a')$ is a successor of $(n, a)$ if and only if $(a, a')$ belongs in $\|\Delta(n, n')\|$.
- The set $\Sigma$ is $N \cup C$.
- The labeling function $L$ is defined such that $L(n, a) = \{n\} \cup \{c \in C \mid a \in \[c]\}$.

Intuitively, this choice of labeling implies that the control nodes are fully observable, while data points are only distinguishable through the constraints in $C$.

Let $V = V'$ denote a special constraint over $V \cup V'$ asserting that the value of each variable in $V$ equals the value of its primed version in $V'$. That is, $V = V'$ is a shorthand for $\bigwedge_{v \in V} v = v'$. For each $n, n' \in N$, we require $\Delta(n, n')$ to have the form

$$
\left( \bigwedge_{V} \alpha_1(V) \land \alpha_2(V') \right) \lor \left( \bigwedge_{V} \beta_1(V) \land V = V' \right)
$$

(8.1)

where $\alpha_1$, $\alpha_2$, and $\beta_1$, are atomic constraints in $C$. Intuitively, we call the first group of disjuncts data-memoryless, because they disallow any relative relationships between the current-state and next-state variables, and call the second group data-invariant, because the variables are not allowed to change their values. Notice that under this restriction, the updates on the data variables are very limited: A data variable cannot change its value relative to its current value, except in the case that every data variable keeps its current value. Although this rules out even a simple update like $v'_1 = v_1 + 10$, we can coarsely approximate certain updates, for example, as $(0 < v_1 \leq 10) \land (10 < v'_1 \leq 20)$.

The key observation is that for any constrained transition system, the equivalence relation induced by the labeling is a bisimulation. Furthermore, the bisimulation is always finite (even if the data domain is infinite).

**Lemma 8.2** Given a constrained transition system $M = (N, N_0, V, D, \Delta, C)$, let $\sim$ be the equivalence relation over $N \times D$ such that for all $(n_1, a_1)$ and $(n_2, a_2)$ in $N \times D$, we have $(n_1, a_1) \sim (n_2, a_2)$ if and only if $L(n_1, a_1) = L(n_2, a_2)$. The relation $\sim$ is a finite bisimulation for $M$. 
Proof: The relation $\sim$ trivially satisfies the first two conditions in the definition of bisimulation. Note that if $(n, a) \sim (m, b)$, by definition, we must have $n = m$ and $a$ and $b$ must satisfy the same set of constraints in $C$. Suppose $(n', a')$ is a successor of $(n, a)$. We need to pick a $d \in D$ with $(n', a') \sim (n', d)$ and $(n', d)$ being a successor of $(n, b)$, as shown in the commutative diagram below (the transition relation is represented by arrows).

\[
\begin{array}{ccc}
(n, a) & \sim & (n, b) \\
\downarrow & & \downarrow \\
(n', a') & \sim & (n', d)
\end{array}
\]

Because $(n', a')$ is a successor of $(n, a)$, by definition, $(a, a')$ must satisfy one of the disjuncts in Formula (8.1).

Case 1: $(a, a') \in [\alpha_1^i (V) \land \alpha_2^j (V')]$ for some $i$. This is equivalent to having $a \in [\alpha_1^i (V)]$ and $a' \in [\alpha_2^j (V')]$. We claim that we can pick $d = a'$. We only need to show that $(n', a')$ is a successor of $(n, b)$. Because $a$ and $b$ satisfy exactly the same set of constraints in $C$, we have in particular $b \in [\alpha_1^i (V)]$. Therefore we have $(b, a') \in [\alpha_1^i (V) \land \alpha_2^j (V')] \subseteq [\Delta (n, n')]$. So $(n', a')$ is a successor of $(n, b)$.

Case 2: $(a, a') \in [\beta^j (V) \land V = V']$ for some $j$. This is equivalent to having $a = a'$ and $a \in [\beta^j (V)]$. We claim that we can pick $d = b$. We clearly have $(n', a) \sim (n', b)$, so we only need to show that $(n', b)$ is a successor of $(n, b)$. Again, since $a$ and $b$ satisfy exactly the same set of constraints in $C$, we have $b \in [\beta^j (V)]$, and therefore $(b, b) \in [\beta^j (V) \land V = V'] \subseteq [\Delta (n, n')]$. So $(n', b)$ is a successor of $(n, b)$.

This completes the proof that $\sim$ is a bisimulation. The bisimulation is finite, because we require $N$ and $C$ to be finite.

### 8.3 Model Checking for Constrained Transition Systems

As a result of Lemmas 8.1 and 8.2, given a constrained transition system and a CTL formula, it is sound and complete to verify the quotient system with respect to $\sim$. Our goal is to encode the quotient system symbolically by a set of Boolean state variables so that BDDs can be used. The control space $N$ is encoded with a finite set $X$ of Boolean variables in the conventional manner. As shown below, the way we handle the data part distinguishes our approach from others. We assume that we have a decision procedure that, given a set of atomic constraints, can determine whether their conjunction is feasible, and, if so, returns a feasible point.

#### 8.3.1 Boolean Encoding of the Quotient System

A set $H \subseteq C$ of atomic constraints defines a set $\kappa(H) \subseteq D$ of data points with

\[
\kappa(H) = [[\bigwedge_{c \in H} c \land \bigwedge_{c \in C \setminus H} \neg c]].
\]

We call each such $\kappa(H)$ a region. Let $C$ be \{\(c_1, c_2, \ldots, c_m\)\}. This suggests a natural Boolean encoding of regions by a set $K$ of $m$ Boolean variables $k_1, k_2, \ldots, k_m$, such that the region $\kappa(H)$ is represented by assigning true to $k_i$ if and only if $c_i \in H$. 
The significance of regions is that they are the building blocks of the data portion of the quotient state space. Given a constrained transition system \((N, N_0, V, D, \Delta, C)\), by definition, the quotient state space with respect to \(\sim\) is \(N \times D^n\), where \(D^n\) is the set of equivalence classes of \(D\) induced by the atomic constraints in \(C\). That is, each \(G \in D^n\) is not empty, and for each \(a_1, a_2 \in G\) and \(c \in C\), we have \(a_1 \in \lceil c \rceil\) iff \(a_2 \in \lceil c \rceil\). In other words, \(D^n\) is precisely the set of all non-empty regions. Note, however, that not every assignment to \(K\) corresponds to an element in \(D^n\), because some assignments represent infeasible constraints and thus empty regions. We define a Boolean function \(Feas(K)\) such that \(Feas(K)\) is true if and only if its encoded constraint is feasible. That is,

\[
Feas(K) = \bigvee_{H \subseteq C | \emptyset \neq H} \left( \bigwedge_{c \in H} c \land \bigwedge_{c \in C \setminus H} \neg c \right). \tag{8.2}
\]

A state in the quotient system is now encoded as an assignment to \(X\) and \(K\), and a set of states can be represented as a Boolean function \(S(X, K) \land Feas(K)\). The quotient transition relation is encoded by a Boolean function

\[
R(X, K, X', K') = Feas(K) \land Feas(K') \land \hat{R}(X, K, X', K') \tag{8.3}
\]

where

\[
\hat{R}(X, K, X', K') = \bigvee_{n_1, n_2 \in N} n_1(X) \land n_2(X') \land \Delta(n_1, n_2)[C \leftarrow K]. \tag{8.4}
\]

Here, the Boolean functions \(n_1(X)\) and \(n_2(X')\) represent the control nodes \(n_1\) and \(n_2\), while the Boolean function \(\Delta(n_1, n_2)[C \leftarrow K]\) is obtained from \(\Delta(n_1, n_2)\) by replacing every \(e_i\) with \(k_i\), every \(e'_i\) with \(k'_i\), and every \(V = V'\) with \(K = K'\) (which is a shorthand for \(\bigwedge_{k_i \in K} k'_i \leftarrow k_i\)).

The BDD for Formula (8.4) is easy to build from the system description, but we also need to construct \(Feas\). Even if the BDD for \(Feas\) is small, in general there may be no efficient way of computing it. The naïve method suggested by Equation 8.2 enumerates all \(2^m\) assignments to \(K\) and invokes the constraint solver to check the feasibility of each case. This method may work well if the number of constraints \(m\) is small.

### 8.3.2 Filtering BDDs

We can avoid building the BDD for \(Feas\) if we have some other way to remove infeasible states. One solution is filtering the functions on the fly: We think of a BDD that represents an arbitrary function \(S\) as a DNF formula whose disjuncts are all the paths from the root to the leaf node 1. The idea of filtering is that, instead of computing \(S \land Feas\), we remove every disjunct \(s\) of \(S\) such that \(s \land Feas\) is unsatisfiable. We denote the resulting function by \(Filter_{Feas} S\). That is, for \(S = \bigvee_i s_i\) where each \(s_i\) is a conjunction of positive or negative versions of a subset of the Boolean variables, we let \(\Pi_S = \{ s_i \mid s_i \land Feas \neq false \}\) and define \(Filter_{Feas} S\) as \(\bigvee \Pi_S\). Note that the function \(Filter_{Feas} S\) depends on the particular DNF representation for \(S\); for BDDs, it depends on the variable order.

Since every disjunct is a conjunction, we can determine whether the disjunct is feasible using the constraint solver without computing \(Feas\). Note also that \(Filter_{Feas} S\) and \(S \land Feas\) are not necessarily the same function. For example, let \(S\) be true, which can also be its
DNF representation. Then, we have $S \land \text{Feas} \equiv \text{Feas}$ but $\text{Filter}_{\text{Feas}} S \equiv \text{true}$. In general, the lemma below holds for any Boolean function $S$.

**Lemma 8.3** We have $S \land \text{Feas} \Rightarrow \text{Filter}_{\text{Feas}} S$.

**Proof:** Observe that $S \land \text{Feas}$ is $\bigvee_i (s_i \land \text{Feas})$. The unsatisfiable disjuncts cannot affect the value of the function and thus can be eliminated. So the function is equivalent to $\bigvee_{s_i \in \Pi_S} (s_i \land \text{Feas})$, which is clearly stronger than $\bigvee \Pi_S$. 

Although $\text{Filter}_{\text{Feas}} S$ still contains some infeasible states, we will show that it is sufficient for model checking.

Recall from Sections 2.2 and 6.4 that the algorithms for symbolic model checking involve four types of operations on sets of states: Boolean operations, satisfiability checking (to detect fixed points), predecessor (or successor) computation, and finding satisfying assignments (for counterexamples). We now see how we perform each of these operations.

The lemma below is trivial to see and implies that for Boolean operations we can delay the removal of infeasible states until the end. (The Boolean functions $S$ and $T$ are arbitrary.)

**Lemma 8.4** We have the following equivalences:

- $(S \land \text{Feas}) \land (T \land \text{Feas}) \equiv (S \land T) \land \text{Feas}$.
- $(S \land \text{Feas}) \lor (T \land \text{Feas}) \equiv (S \lor T) \land \text{Feas}$.
- $(\neg(S \land \text{Feas})) \land \text{Feas} \equiv (\neg S) \land \text{Feas}$.

**Proof:** By simple Boolean manipulations.

The functions on the left hand side are the straightforward way of doing the operations. On the right hand side, we do the same operations but remove infeasible states only in the final result. The next lemma implies that if we only care whether the function is satisfiable, then even the final result does not need to be conjoined with $\text{Feas}$; instead, we can check the satisfiability of the filtered result.

**Lemma 8.5** $S \land \text{Feas}$ is satisfiable if and only if $\text{Filter}_{\text{Feas}} S$ is satisfiable.

**Proof:** The only-if direction follows from Lemma 8.3. Conversely, let $\bigvee_i s_i$ be our DNF representation for $S$. If $S \land \text{Feas}$, or equivalently, $\bigvee_i (s_i \land \text{Feas})$, is not satisfiable, then by definition the set $\Pi_S$ is empty, and therefore $\text{Filter}_{\text{Feas}} S$ is false.

The next lemma implies a way of computing predecessors and successors:

**Lemma 8.6** For any disjoint sets $Z$ and $K$ of Boolean variables and any Boolean function $T(Z, K)$, we have the following equivalence:

$$\exists K. (\text{Feas}(K) \land T(Z, K)) \equiv \exists K. \text{Filter}_{\text{Feas}(K)} T(Z, K).$$
**Proof:** The forward direction follows from Lemma 8.3 and the monotonicity of existential quantification. For the other direction, consider any assignment $A_Z$ to $Z$ that satisfies the right hand side. We want to show that it satisfies the left hand side as well. From the formula, there exists an assignment $K$ that together with $A_Z$ satisfies $\Filter_{\Feas[K]} T(Z, K)$. Let our DNF representation for $T(Z, K)$ be $\bigvee_i (s_i(Z) \land t_i(K))$. Therefore, the assignments satisfy one of the disjuncts $s_j(Z) \land t_j(K)$, and by the definition of filtering, the function $t_j(K) \land \Feas(K)$ is satisfiable. Let $A_K$ be an assignment to $K$ that satisfies $t_j(K) \land \Feas(K)$. We have that $A_Z$ and $A_K$ together satisfy $s_j(Z) \land t_j(K) \land \Feas(K)$. Note that this is one of the disjuncts in the DNF representation for $\Feas(K) \land T(Z, K)$, which can be written as $\bigvee_i (s_i(Z) \land t_i(K) \land \Feas(K))$. So, $A_Z$ and $A_K$ satisfy $\Feas(K) \land T(Z, K)$ as well, and thus $A_Z$ satisfies the left hand side of the equivalence. \qed

Observe that by Formulas (8.1) and (8.4), the function $\hat{R}(X, K, X', K')$ can be rewritten as

$$R_M(X, K, X', K') \lor (R_1(X, K, X') \land K = K')$$

for some Boolean functions $R_M$ and $R_1$, which represent the data-memoryless and data-invariant portions of the transition relation respectively. The next lemma gives a way of computing the successors of a set of states without using $\Feas$ (predecessor computation is similar).

**Lemma 8.7** We have the following equivalence:

$$\exists X. \exists K. \left( (S(X, K) \land R(X, K, X', K')) \equiv (U_M(X', K') \lor U_1(X', K')) \land \Feas(K') \right)$$

where

$$U_M(X', K') = \exists X. \exists K. \Filter_{\Feas[K]} (S(X, K) \land R_M(X, K, X', K'))$$

$$U_1(X', K') = \exists X. \Filter_{\Feas[K]} (S(X, K) \land R_1(X, K, X')).$$

**Proof:** By expanding out the definition of $R$ and distributing the existential quantifications over disjunctions, the left hand side is equivalent to

$$\left( \exists X. \exists K. \left( \Feas(K) \land S(X, K) \land R_M(X, K, X', K') \right) \lor \exists X. \exists K. \left( \Feas(K) \land S(X, K) \land R_1(X, K, X') \land K = K' \right) \land \Feas(K') \right),$$

which by Lemma 8.6 is equivalent to $(U_M(X', K') \lor V_1(X', K')) \land \Feas(K')$, where

$$V_1(X', K') = \exists X. \exists K. \Filter_{\Feas[K]} (S(X, K) \land R_1(X, K, X') \land K = K')$$

$$\equiv \exists X. \exists K. \left( \Filter_{\Feas[K]} (S(X, K) \land R_1(X, K, X')) \land K = K' \right).$$

We could pull out $K = K'$ from the scope of the filtering operation because it does not constrain $K$ and hence cannot affect the satisfiability of the disjuncts. The last function above can be easily seen to be equivalent to $U_1(X', K')$. \qed
\begin{align*}
\text{FILTER}(B; \text{BDD}): \text{BDD} \\
\text{LABEL}(B,\text{true}) \\
\text{return PRUNE}(B)
\end{align*}

\begin{align*}
\text{PRUNE}(B; \text{BDD}): \text{BDD} \\
&\text{if } B = 0 \text{ or } B = 1 \text{ then return } B \\
&\text{let } y_j = B.\text{var} \\
&\text{if } j > l \text{ then return } B' \\
&\text{if } \langle B, B' \rangle \text{ is in cache, return } B' \\
&\text{if } B.\text{edge}_0 = \top \\
&\quad \text{then } B_0 \leftarrow \text{PRUNE}(B.\text{child}_0) \\
&\quad \text{else } B_0 \leftarrow 0 \\
&\text{if } B.\text{edge}_1 = \top \\
&\quad \text{then } B_1 \leftarrow \text{PRUNE}(B.\text{child}_1) \\
&\quad \text{else } B_1 \leftarrow 0 \\
&\quad B' \leftarrow \text{ITE}(B.\text{var}, B_0, B_1) \\
&\quad \text{insert } \langle B, B' \rangle \text{ and } \langle B', B' \rangle \text{ in cache} \\
&\text{return } B'
\end{align*}

\begin{align*}
\text{LABEL}(B; \text{BDD, } \alpha; \text{Constraint}): \{ \top, \bot \} \\
&\text{if } B = 0 \text{ then return } \bot \\
&\text{if } B = 1 \text{ then return } \mathcal{FAS}(\alpha) \\
&\text{let } y_j = B.\text{var} \\
&\text{case} \\
&\quad j < u: \ldots \ldots \text{(Case 1: upper layer)} \\
&\qquad \text{if } B.\text{edge}_0 = ? \text{ then} \\
&\qquad \qquad r_0 \leftarrow \text{LABEL}(B.\text{child}_0, \alpha) \\
&\qquad \qquad B.\text{edge}_0 \leftarrow r_0 \\
&\qquad \text{else } r_0 \leftarrow B.\text{edge}_0 \\
&\qquad \text{if } B.\text{edge}_1 = ? \text{ then} \\
&\qquad \qquad r_1 \leftarrow \text{LABEL}(B.\text{child}_1, \alpha) \\
&\qquad \qquad B.\text{edge}_1 \leftarrow r_1 \\
&\qquad \text{else } r_1 \leftarrow B.\text{edge}_1 \\
&\quad u \leq j \leq l: \ldots \text{(Case 2: middle layer)} \\
&\qquad \text{if } r_0 = \top \text{ then } B.\text{edge}_0 \leftarrow \top \\
&\qquad \quad r_1 \leftarrow \text{LABEL}(B.\text{child}_0, \alpha \land \lnot y_j) \\
&\qquad \text{if } r_0 = \top \text{ then } B.\text{edge}_1 \leftarrow \top \\
&\quad j > l: \ldots \ldots \text{(Case 3: lower layer)} \\
&\quad \text{return } \mathcal{FAS}(\alpha) \\
&\text{endcase} \\
&\text{if } r_0 = \top \text{ or } r_1 = \top \text{ then return } \top \\
&\text{else return } \bot
\end{align*}

Figure 8.1: A BDD filtering algorithm

The use of $U_1$ instead of $V_1$ is pragmatic. In general, the cost of filtering depends on the number of disjuncts, and unfortunately there are necessarily $2^m$ disjuncts for the function $K = K'$. Using $U_1$ avoids this exponential cost by eliminating $K = K'$ from the representation.

As a result of the lemmas above, the only necessary change to the conventional symbolic fixed-point algorithms is to use the right hand sides of Lemmas 8.5 and 8.7 to detect fixed points and compute successors respectively.

Finally, to find a feasible state in $S$ for generating counterexamples, we compute Filter\text{\textit{Feas}}\text{\textit{S}} and pick an arbitrary disjunct $s$, which corresponds to a partial assignment to the variables. The unassigned variables not in $K$ can be set arbitrarily. We then use the constraint solver to find a feasible point satisfying $s$, and evaluate every atomic constraint in $C$ at the feasible point, thereby obtaining a complete assignment to $K$.

8.3.3 A Filtering Algorithm

Filtering a BDD amounts to removing all paths from the root to the leaf 1 that correspond to infeasible constraints. Figure 8.1 shows a BDD filtering algorithm Filter. We assume that the function is to be filtered with respect to \textit{Feas}(K); our algorithm can be trivially changed for filtering with respect to \textit{Feas}(K'). The algorithm consists of two phases: In
the labeling phase, it labels the edges along all feasible paths with \( \top \), and in the pruning phase, it redirects the edges not labeled with \( \top \) to the leaf 0.

Each non-leaf BDD node has five fields. The \( \text{var} \) field stores the BDD variable. The \( \text{child}_0 \) field points to the 0-child BDD. The \( \text{edge}_0 \) field is the label of the 0-edge, which is either \( \top \) (feasible), \( \bot \) (infeasible), or ? (unknown, the initial value). The \( \text{child}_1 \) and \( \text{edge}_1 \) fields are symmetric. Suppose the BDD variables in order are \( y_1, y_2, \ldots, y_n, \ldots, y_K, \ldots, y_p \), where \( y_n \) and \( y_1 \) are the first and last variables in \( K \). That is, we have \( y_n, y_i \in K \) and \( y_i \notin K \) for \( i \in \{ y_1, \ldots, y_{n-1} \} \) and \( i \in \{ y_{n+1}, \ldots, y_p \} \). We call the part of the BDD with variables \( y_1 \) through \( y_{n-1} \) the upper layer, \( y_n \) through \( y_t \) the middle layer, and \( y_{t+1} \) through \( y_p \) the lower layer. Therefore, only the middle layer contains variables in \( K \).

The routine \textsc{Label} traverses the paths in a depth-first manner, keeping track of the corresponding constraint \( \alpha \) as it walks down a path. Case 2 is important for correctness, while Cases 1 and 3 are for optimizations—each node in the upper layer is not visited more than once (Case 1), and nodes in the lower layer are not explored at all (Case 3). The constraint solver \( \mathcal{FES} \) takes a constraint \( \alpha \), and returns \( \top \) if \( \alpha \) is feasible, or \( \bot \) otherwise. The function \( \mathcal{I} \) “interprets” the BDD variables as data constraints. For each \( x \) in \( X \), we have \( \mathcal{I}(\neg x) = \mathcal{I}(x) = \text{true} \), and for each \( k_i \) in \( K \), we have \( \mathcal{I}(k_i) = c_i \) and \( \mathcal{I}(\neg k_i) = \neg c_i \). The routine \textsc{Prune} performs the pruning phase. The function \textsc{ITE} takes a BDD variable \( y \) and two BDDs \( B_0 \) and \( B_1 \), and returns a BDD with top variable \( y \), 0-child \( B_0 \) and 1-child \( B_1 \).

Assuming that \( \mathcal{FES} \) takes constant time, the time complexity of \textsc{Filter} is linear in the number of nodes in the upper layer, and in the number of paths in the middle layer. The latter is the major bottleneck of the algorithm, as the number of paths can be exponential in the number nodes in the worst case.

### 8.4 Example

We implemented the above algorithms in SMV. The constraint solver used was \textsc{QuadCLP}(\( R \)) [118], a less incomplete solver than \textsc{CLP}(\( R \)) for quadratic constraints. We had access only to the executable of the solver, so it was integrated with SMV through inter-process communication.

We illustrate our technique with a simple statecharts-like system shown in Figure 8.2. It is a hypothetical automobile cruise-control system with collision avoidance. The idea is that when the automobile is too close to the vehicle in front, the cruise-control system will automatically deactivate itself.

Three numeric inputs to the system are \( s_0 \), the velocity of the vehicle; \( s_f \), the velocity of the front vehicle; and \( d \geq 0 \), the distance between the vehicles. (In reality, \( s_f \) may be estimated from the current and previous values of \( s_0 \) and \( d \).) The closeness of the two vehicles is based on time rather than distance. Let \( s \) be \( s_0 - s_f \). The estimated time to collision is \( d/s \). If this quantity is less than some threshold \( t \), the two vehicles may be considered too close. However, if \( s \) is positive but small, the two vehicles can get very close without triggering the condition. To fix this problem, the following condition can be used instead:

\[
\frac{d - K/d}{\max(\varepsilon, s)} < t.
\]
Figure 8.2: A hypothetical automobile cruise-control system with collision avoidance

The max function is for avoiding division-by-zero. Subtracting $K/d$ from the numerator makes the inequality true when $d$ is tiny, regardless of the value of $s$. The positive value $K$ depends on the “sensitivity level” (large $K$ for high sensitivity). Although this example is naive, the inequality is exactly the one used in TCAS II for threat detection. We arbitrarily chose $\epsilon = 1$ and $t = 2$.

As shown in Figure 8.2, the system is divided into four parallel state machines. We omit the events from the diagram, but assume the machines are synchronized in an oblivious manner: In the first microstep, Own-Vel and Front-Vel execute concurrently; in the second and third microsteps, Sensitivity and Cruise-Control execute respectively. Because the values of all the inputs are nondeterministic at the beginning of a macrostep, but during a macrostep they are assumed to be unchanged by the synchrony hypothesis, this system can be modeled as a constrained transition system: Microstep transitions are data-invariant while transitions across macrostep are data-memoryless.

We verified several invariants of this model using our prototype implementation. In the model, there are (at least) six Boolean variables representing constraints: $s_0 \geq 25$, $s_0 < 15$, $s_f \geq 25$, $s_f < 15$, $((d - 1)/d)\max(1, s_0 - s_f) < 2$, and $((d - 10)/d)\max(1, s_0 - s_f) < 2$. Additional Boolean variables are used when the property being verified contains other constraints. We focused on verifying that Cruise-Control is never in Cruise under certain conditions, for example, when $d$ is less than 2. That is, in CTL,

$$\text{AG} \neg (d < 2 \land \text{Cruise-Control} = \text{Cruise}).$$

This property is false because the two transitions into Cruise are not guarded by $\neg$too-close. The model checker correctly showed a counterexample. After strengthening the guards on
the two transitions, the property was verified true. Two other related properties were also successfully verified: Cruise-Control is never in Cruise when either (1) d is less than 4 and Sensitivity is High, or (2) d is less than 20, Sensitivity is High, and Sen is Low. Each of the above properties was evaluated within a second by our prototype implementation. The numbers of calls made to the constraint solver were at most about 30% of the number of calls required to construct Feas.

8.5 Discussion and Related Work

In this chapter, we have introduced a symbolic model checking technique for systems with possibly nonlinear arithmetic constraints on the state transitions. The key idea is to use a constraint solver to identify infeasible combinations of constraints, which are then pruned from the BDDs. The specific contributions are the development of a method that is sound and complete for a class of systems, along with a proof-of-concept implementation that shows that the method works for a small example.

We have opted to augment BDD-based model checking to deal with nonlinear constraints. The main reason is that we are interested in systems with large and complex control logic, for which only BDD-based model checking has proven to work well. The high dependence between control and data paths also prevents us from separating them for verification, a technique that is sometimes used in microprocessor verification [29].

Most work on handling nonlinearity in verification has been focused on arithmetic circuits. One approach is to use BMDs or *BMDs [25] and their variants, such as HDDs [43]. Although they can represent the product xy concisely, representing the constraint xy ≥ z still requires exponential size. In fact, it has been proved that many generalizations of these representations cannot solve the problem [133]. Our approach can deal with not only integral multiplicative constraints but also arbitrarily complex (e.g., trigonometric) constraints over finite or infinite domains, provided an appropriate constraint solver is available.

Recently, Boolean satisfiability solvers in lieu of BDDs are used for symbolic model checking Biere et al. [15]. There are examples, including a hardware multiplier, in which the BDDs blow up but the Boolean solver works well. It would be interesting to see whether the technique is also effective in our domain.

Abstracting a constraint as a single Boolean variable is not a new idea [e.g., 51, 52]. However, since infeasible combinations of constraints are not automatically detected, either the approach is incomplete, or it requires substantial manual abstraction. Wang et al. [135] also represent certain timing constraints in distributed real-time systems as BDD variables. However, to ensure soundness and completeness, their method requires building a BDD in exponential time before running the fixed-point algorithm. We try to avoid similar preprocessing by restricting the class of systems that we deal with and by pruning the BDDs on the fly.

We only describe the basic algorithms in this chapter, and there are many possibilities for optimizations. For examples, for many properties, it is likely that we do not need to interpret every atomic constraint. We can conservatively treat some of them as control variables to reduce the number of constraints that the solver needs to handle. Another observation is that a lot of our atomic constraints involve comparing a variable with constants, e.g.,
$v_1 < 10$, $10 \leq v_1 < 20$, $20 \leq v_1 < 30$, and $30 \leq v_1$. In these cases, we can reduce the number of Boolean variables in $K$ by partitioning the domain of the variable and using a binary encoding for the partitions. For the example above, instead of having four Boolean variables, we will have two Boolean variables to encode the four intervals. However, this scheme would make interpreting the Boolean variables in the filtering algorithm harder. We also note that although generally solving nonlinear constraints remains a hard problem, most of our constraints have low polynomial degrees and are sparse (that is, each atomic constraint depends on only a small subset of data variables). It may be possible to exploit such special structures to improve the performance of the solver.

Our technique can be generalized in various ways. The idea can be applied to systems with transitions annotated by assertions in any theory, if a decision procedure for the theory is available. Allowing transitions that are not data-memoryless or data-invariant can make the technique more useful, but that would probably require computing a bisimulation before applying model checking, or approximating one on the fly. Doing so is likely to blow up the number of BDD variables required. There are also many open questions. For example, the choice of variable order needs investigation because it affects both the BDD size and the number of paths traversed. We also need to experiment with larger systems like TCAS II to see whether the technique is practical. Although TCAS II does not fall into our class of constrained transition systems because environmental inputs are remembered using the Prev construct, the restricted model should still enable us to reason about some interesting properties. At the very least, we can conservatively analyze behaviors within a single macrostep.

Note that the work on nonlinear hybrid systems [82] differs from ours because nonlinearity there refers to constraints over the derivatives of the variables.
Lessons Learned and Future Work

What can we learn from our experience of translating the state-machine specifications to SMV (Chapter 3), model-checking the specifications (Chapters 4 and 5), developing and using the optimization techniques (Chapters 6 and 7), and extending model checking with constraint solving (Chapter 8)? This chapter summarizes some issues and points to some directions for future research.

Finite-state models One concern that many software researchers have is that BDD-based model checking can only apply to finite-state systems, but software often has an infinite number of states. Nevertheless, many specifications for safety-critical process-control software, such as the EPD model in Chapter 5, are, in fact, finite-state. In addition, a current research trend is to develop symbolic representations and model-checking algorithms to directly verify some classes of infinite-state systems [4, 17, 26]; our approach of constraint solving in Chapter 8 can be considered one of them. These techniques, however, are far less mature than purely BDD-based methods. Fortunately, many infinite-state systems can be abstracted as finite-state ones, which are then amenable to conventional model checking [99]. Often, the abstraction is conservative in the sense that, if the properties hold in the abstraction, then they are guaranteed to hold in the full specification. If the goal is to find errors instead of proving correctness, this preservation guarantee can be forsaken, using techniques like model checking to find counterexamples but not to guarantee properties [100]. In our work, for example, some inputs of one of the versions of TCAS II are specified as real numbers, which were discretized as integers in our model (Section 4.2.2). The counterexamples we found in the finite-state model also exist in the full specification.

Perhaps the real problem in practice is not the lack of a finite-state model, but rather, the lack of any formal model at all. The use of formal modeling for software, safety-critical or not, is still not the norm. Given the widely recognized needs for building abstract models in other engineering disciplines, the deficiency of formal models in software development may seem an anomaly. The resistance can perhaps be attributed to a fundamental difference between models of software and models of other artifacts: A model of, say, a bridge is not a bridge, but, in practice, a model of a piece of software is another piece of software! It is therefore hard to convince developers to build two pieces of software for the same job. The lack of any necessary connection between the specification and the implementation further aggravates the problem—properties proved about the former may not carry over to the latter.
Fortunately, techniques to bridge the gap between specification and implementation are emerging, and their success seems important for demonstrating the cost-effectiveness of formal modeling, a requisite for applying model checking. Ideally, we would like to generate code automatically from the specification. The approach, albeit difficult in general, may well be feasible in our restricted domain.  Heimdahl and Keenan [76] automatically derive C++ code from RSML specifications, but report that the generated code is an order of magnitude slower than and twice as large as hand-optimized code. However, the inefficiency does not seem fundamental, and further optimizing the generated code should be possible. Or, one can statically check the consistency between the specification and a hand-coded implementation [38]. Recent work has also proposed to systematically generate test cases from the specification using model checkers' ability to generate counterexamples [8, 58, 62].

A radical alternative is to bypass the specification altogether and analyze the code using model checking. Recent research has started to attack this problem by either automatically extracting a finite-state model from user-annotated code [87], or by running the code systematically as if performing model checking [65]. The former has the advantage of being able to analyze an incomplete program, while the latter explores the actual program instead of an abstraction. Initial results show their promise in catching concurrency bugs in communications software [66, 88]. For analyzing code, there may also be techniques we can borrow from data-flow analysis, which is closely related to symbolic model checking [128, 132].

**Temporal Properties** Apart from a state-machine model, another formal entity needed for model checking is a temporal property. Critics of model checking for software often point out the difficulty in identifying properties to check. Although the problem does exist, it seems unlikely to be a major hurdle. After all, writing test cases is nontrivial too, but few would abandon testing simply because of this. Indeed, identifying temporal properties resembles finding test cases in many ways. One can regard a temporal property as a specification of a suite of test cases: Instead of enumerating inputs and the corresponding outputs, we make the intents of the test cases evident by describing the desirable relationships between the inputs and the outputs in temporal logic.

A number of other approaches can also be used: For safety-critical systems, Jaffe et al. [101] described many properties that should be satisfied by requirements specifications of safety-critical systems, such as determinism, completeness, etc. We also believe that certain domain-specific properties like output agreement (Section 4.3.5 on page 42) are applicable across many domains. Some other analysis problems, such as deviation analysis [127], can be posed as model-checking problems. In addition, candidate properties may arise in the field. For example, pilots have reported anomalous behavior that they observed while using some versions of TCAS II. Such anomalies could be checked against the specification, and one may determine if the problem is in the specification, in the implementation, or in the report itself. An alternative approach is to identify certain common patterns of temporal-logic formulas [55]. Not only can they help the user formally express the property, these patterns can also suggest additional properties to check.

Note that, to disclose a particular flaw, it is usually not necessarily to check a specific property. In other words, a bug often manifests itself in the counterexamples to a number
of different properties [112]. For example, the bug B1 in Section 5.3 was revealed in several different analyses. Nevertheless, finding a set of properties with good coverage does increase our confidence in the correctness. In hardware verification, some coverage metrics have recently been proposed [89], and it would be interesting to use or adapt them for our purposes.

As opposed to merely coming up with properties and checking them, an interesting research direction is to infer properties, such as invariants [102, 117]. We have started to investigate the inference of more general temporal-logic properties using a technique similar to symbolic model checking [33].

Scale As pointed out in Chapter 1, another prevalent concern is software's apparent lack of "regularity" for symbolic model checking. On the contrary, our experience has shown that BDDs can capture the complex control structures of TCAS II and the EPD system, provided we pay attention to the reasons for BDD blowups. To make the technique scale to even larger state-machine specifications, we believe that it is important to design the language, the algorithms, and the tools with symbolic model checking in mind.

Consider first the specification language. The notion that the design of a language determines how efficient the analysis can be done is well accepted in the area of programming languages. Specification languages are not different in this regard. Our experience in this work has identified some factors that can affect the efficiency of symbolic traversals, and we hope that the insights will lead to languages that are more amenable to symbolic model checking. In addition, a particularly undesirable feature of both RSML and statecharts is their lack of compositionality for verification: In general, a temporal property proved locally about a state machine may not hold in the global system (because, for example, of the event broadcast mechanism and of the state hierarchy). This makes compositional analysis nontrivial. Rectifying this problem at the language level would be valuable.

Our optimization techniques have shown that the model-checking algorithms can be improved dramatically if the high-level state-machine language is taken into account. In other words, using a purely syntax-directed translation and relying on the model checker to perform optimizations would not be efficient. Static analysis can be used to facilitate the construction of a model that is more amenable to model checking (e.g., Chapter 6) [see also 47]. We argue that other software researchers should also try to obtain a thorough understanding of the algorithms used by the model checker. This is especially critical for symbolic techniques, whose performance can be very sensitive to the model, the property, and the various parameters chosen. Otherwise, discouraging outcomes may result simply from the artifacts of the model or the tool, which has little to do with the technique itself.

In our work, we translated the high-level models into inputs to an existing model checker, which we had modified to incorporate our optimizations. Often we needed to modify the input language of the model checker as well (for example, because an optimization needed to know which Boolean variables encoded events). This created an unnecessary extra level of indirection. A more direct approach is to implement the model-checking algorithms in the CASE tool itself using one of the BDD libraries available. We believe that the availability of model-checking libraries, instead of just BDD libraries, would further facilitate the tight integration.
It is important to design tools that domain experts feel comfortable in using. For example, AND/OR tables (Section 3.1) were designed to replace propositional logic for specifying guarding conditions in RSML because aircraft engineers did not find the latter natural [107]. Similarly, domain experts may not like temporal logic or understand its intricacies. Finding intuitive alternatives (perhaps even sacrificing some of the expressive power of the logic) is critical for gaining wider acceptance. For better scaling, the tool should also allow the user to easily specify an abstraction of the model for verification. Although automatic abstraction techniques (Section 7.2) helps, insights from the user would undoubtedly enable more significant reduction. The user should be able to specify overapproximation (e.g., by projection or nondeterminism) or underapproximation (e.g., by specialization, a la program slicing [136]).

In spite of the progress we have made, automatic formal verification of complex systems will continue to be an intractable problem. Our strategy has been to concentrate on real systems that practitioners care about, and try to come up with techniques that can address important classes of problems. We also believe that a lot of leverage can be gained if software researchers can get a deeper understanding of the underlying algorithms in the model checkers that they use, while verification researchers pay more attention to how their techniques are employed in practice.
Appendix A

Program Listings

A.1 SMV Program for the Translation Example

This is the complete SMV translation of the statecharts example in Figure 3.1 on page 14 (but as explained in Section 4.2.3, SMV does not handle this program well as it is).

 MODULE main
 VAR
   u: boolean;
   v: boolean;
   w: boolean;
 switch: {up, down, test};
 alt: 0..20000;
 prev-alt: 0..20000;
 Alt-Layer: {High, Mid, Low};
 Alarm: {Shutdown, Operating};
 Mode: {Off, On};
 Volume: {1, 2};
 time-Mid: 0..5;
 DEFINE
  stable := !(u|v|w);
 in-Sys := 1;
 in-Alt-Layer := in-Sys;
 in-High := in-Alt-Layer & Alt-Layer = High;
 in-Mid := in-Alt-Layer & Alt-Layer = Mid;
 in-Low := in-Alt-Layer & Alt-Layer = Low;
 in-Alarm := in-Sys;
 in-Shutdown := in-Alarm & Alarm = Shutdown;
 in-Operating := in-Alarm & Alarm = Operating;
 in-Mode := in-Operating;
 in-Volume := in-Operating;
 in-Off := in-Mode & Mode = Off;
 in-On := in-Mode & Mode = On;
 in-1 := in-Volume & Volume = 1;
 in-2 := in-Volume & Volume = 2;
 t1 := in-High & u & alt >= 9950;
 t2 := in-Mid & u
   & 1950 <= alt & alt <= 10050;
t3 := in-Low & u & alt <= 2050;
t4 := in-Mid & u & alt > 10050;
t5 := in-High & u & alt < 9950;
t6 := in-Low & u & alt > 2050;
t7 := in-Mid & u & alt < 1950;
t8 := in-Shutdown & u & switch=up;
t9 := in-Shutdown & u & switch=down;
t10 := in-Off & w & c;
t11 := in-On & w & in-Mid;
t12 := in-1 & v;
t13 := in-2 & v;
t14 := in-Shutdown & u & switch=test;
c := in-Low &
   (alt<1000
    | (alt<1500 & prev-alt<1500)
    | time-Mid >= 5);

ASSIGN
init(Alt-Layer) := Mid;
next(Alt-Layer) :=
case
  t1|t4 : High;
  t2|t5|t6 : Mid;
  t3|t7 : Low;
  1 : Alt-Layer;
esac;
init(Alarm) := Shutdown;
next(Alarm) :=
case
  t8|t14 : Operating;
  t9 : Shutdown;
  1 : Alarm;
esac;
init(Mode) := Off;
next(Mode) :=
case
  t10|t14 : On;
  t8|t11 : Off;
  1 : Mode;
esac;
init(Volume) := 1;
next(Volume) :=
case
  t8|t13|t14 : 1;
  t12 : 2;
  1 : Volume;
esac;
init(w) := 0;
next(w) := t1|t2|t3|t4|t5|t6|t7;
next(u) :=
stable: {0,1};
1 : 0;
esac;
next(v) :=
case
  stable: {0,1};
  1 : 0;
esac;
next(switch) :=
case
  stable & !next(stable): {up, down, test};
  1 : switch;
esac;
next(alt) :=
case
  stable & !next(stable): 0..20000;
  1 : alt;
esac;
next(prev-alt) :=
case
  stable: alt;
  1 : prev-alt;
esac;
next(time-Mid) :=
case
  t2|t4|t7 : 0;
  stable & time-Mid < 5: time-Mid + 1;
  1 : time-Mid;
esac;

A.2 SMV Programs for the Experiments on the Parameterized Examples

This section gives the SMV programs for the experiments on the parameterized examples summarized in Table 6.2 on page 62. They show the differences in the efficiency of analyzing oblivious and non-oblivious models, as well as the effects of pruning using mutually exclusive events and microstep counters.

The following shows the programs for the non-oblivious models in Figure 6.1 without any optimization. The expressions $t_i^0$ and $t_i^1$ represent respectively the conditions under which the transitions to states 0 and 1 in machine $A_i$ are enabled. The parameter $n$ is the number of state machines.

```
MODULE main
DEFINE stable := !x0 & !x1 & ... & !xn;
VAR
  x0 : boolean;
ASSIGN
```
next(x₀) := case
  stable: {0,1};
  1: 0;
  esac;

For each i with 1 ≤ i ≤ n:
DEFINE
  t₁ᵢ := xᵢ₋₁ & aᵢ = 0 & cᵢ;
  t₀ᵢ := xᵢ₋₁ & aᵢ = 1 & !cᵢ;

VAR
  cᵢ: boolean;
  aᵢ: boolean;
  xᵢ: boolean;

ASSIGN
  init(aᵢ) := 0;
  next(aᵢ) := case
    t₀ᵢ: 0;
    t₁ᵢ: 1;
    1: aᵢ;
  esac;
  next(cᵢ) := case
    stable: {0,1};
    1: 0;
    esac;
  init(xᵢ) := 0;
  next(xᵢ) := t₁ᵢ | t₀ᵢ;

SPEC
  AG !(stable & a₋₁ = 0 & aₙ = 1)

Other models are obtained by modifying the code above.

- For the oblivious models in Figure 6.2:
  - The definitions of t₁ᵢ and t₀ᵢ are changed to the following.

    DEFINE
    t₁ᵢ := x₀ & c₁;
    t₀ᵢ := x₀ & !c₁;

    For each i with 1 < i ≤ n:
    t₁ᵢ := x₋₁ & ((aᵢ = 0 & p₋₁ᵢ = 0 & a₋₁ᵢ = 1 & cᵢ) | (aᵢ = 1 & (p₋₁ᵢ = 0 | a₋₁ᵢ = 1 | cᵢ)));
    t₀ᵢ := x₋₁ & ((aᵢ = 1 & p₋₁ᵢ = 1 & a₋₁ᵢ = 0 & !cᵢ) | (aᵢ = 0 & (p₋₁ᵢ = 1 | a₋₁ᵢ = 0 | !cᵢ)));

    - The following code is added. The variable p₋₁ᵢ encodes the previous state of machine Aᵢ.

      For each i with 1 ≤ i < n:
      VAR
      p₋₁ᵢ: boolean;
      ASSIGN
      init(p₋₁ᵢ) := 0;
      next(p₋₁ᵢ) := case
        stable: aᵢ;
1: pa_i;
esac;

• For pruning using mutually exclusive events (Section 6.1), the following code is added:

\[
\begin{align*}
\text{TRANS} & \neg (x_0 \& x_1) \\
\text{TRANS} & \neg (x_0 \& x_2) \\
\vdots \\
\text{TRANS} & \neg (x_0 \& x_n) \\
\text{TRANS} & \neg (x_1 \& x_2) \\
\vdots \\
\text{TRANS} & \neg (x_{n-1} \& x_n)
\end{align*}
\]

• For microstep counters (Section 6.3):
  
  - The following code is added

\[
\begin{align*}
\text{VAR} \\
\text{mc} & : 0..n; \\
\text{ASSIGN} \\
\text{init}(\text{mc}) & := x_0; \\
\text{next}(\text{mc}) & := \text{case} \\
\text{mc} = 0 & : \text{next}(x_0); \\
1 & : (\text{mc} + 1) \mod (n + 1); \\
esac;
\end{align*}
\]

  - The expression stable is redefined as

\[
\text{DEFINE} \quad \text{stable} := \text{mc} = 0;
\]

  - An extra conjunct \text{mc} = i is added to the definition of every \(t_i^0\) and \(t_i^1\).

The variable order used was \((\text{mc}, x_0, c_1, a_1, (pa_1), x_1, c_2, a_2, (pa_2), \ldots, x_n, \text{where mc was used only in the models with a microstep counter, and the \(pa_i\)'s were used only in the oblivious models.} \)
Bibliography


Bibliography


