Optimizing Symbolic Model Checking for Statecharts

William Chan, Richard J. Anderson, Paul Beame, David H. Jones, David Notkin, *Senior Member*, *IEEE*, and William E. Warner

Abstract—Symbolic model checking based on binary decision diagrams is a powerful formal verification technique for reactive systems. In this paper, we present various optimizations for improving the time and space efficiency of symbolic model checking for systems specified as statecharts. We used these techniques in our analyses of the models of a collision avoidance system and a fault-tolerant electrical power distribution (EPD) system, both used on commercial aircraft. The techniques together reduce the time and space requirements by orders of magnitude, making feasible some analysis that was previously intractable. We also elaborate on the results of verifying the EPD model. The analysis disclosed subtle modeling and logical flaws not found by simulation.

Index Terms—Formal verification, symbolic model checking, binary decision diagrams, requirements specifications, statecharts, RSML, TCAS II, partitioned transition relation, automatic abstraction, fault tolerance, avionic systems.

1 Introduction

Formal verification based on state exploration can be considered an extreme form of simulation or testing: *Every* possible behavior of the system is checked for correctness. Symbolic model checking [1] based on binary decision diagrams (BDDs) [2] is an efficient state-exploration technique for finite-state systems. It has been successful in verifying and in detecting faults in many industry-scale hardware systems. Its application to nontrivial software or process-control systems is far less mature, but is increasingly promising. For example, we obtained encouraging results from applying symbolic model checking to a portion of a preliminary version of the system requirements specification of TCAS II, a complex system for mid-air collision avoidance [3]. The full requirements, comprising about 400 pages, are written in the Requirements State Machine Language (RSML) [4], a state-machine language based on statecharts [5]. In this article, we report on another case study of applying symbolic model checking to a statecharts model, that of an electrical power distribution (EPD) system for Boeing aircraft, and describe in detail various techniques for optimizing the analyses in these two studies.

By representing state sets and relations implicitly as BDDs for symbolic model checking, the sheer number of reachable states is no longer the obstacle to analysis. Instead, the limitation is the size of the BDDs, which depends on the structure of the system analyzed. Considerable efforts at formal verification of hardware have been focused on controlling the BDD size for typical circuits. However, transferring this technology to new domains may require alternative techniques and heuristics to combat the BDD-blowup problem. In this work, we develop some intuitions about some reasons for BDD blow-ups. We modify the models and the model checker and use a simple abstraction technique to improve the time and space efficiency of the TCAS II and the EPD analyses. Experimental results show orders-of-magnitude reduction in the time and space requirements. These improvements have made feasible some analysis that was previously intractable.

The specific techniques that we will discuss are:

- *Managing forward and backward traversals* to reduce the *size of the BDD* generated at each search iteration. Notably, we find that forward traversals are much less efficient for our models than backward ones (Section 4.4) and we further improve backward traversals by making certain invariants (in particular, that some events are mutually exclusive) explicit in the search (Section 4.1).
- *Semantics-preserving transformation* of the model to again reduce the size of the BDDs generated. We identify certain styles for synchronization in statecharts that are more efficient for symbolic model checking (Section 4.2). For statecharts not written in these styles, we give procedures to automatically modify their internal representations to greatly improve the performance of their analysis. This is achieved by transparently incorporating a so-called *microstep counter* into the statecharts to take over the synchronization (Section 4.3).
- More sophisticated *conjunctive partitioning* of the transition relation and applying *disjunctive*

- R.J. Anderson, P. Beame, and D. Notkin are with the Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350. E-mail: {anderson, beame, notkin}@cs.washington.edu.
- D.H. Jones and W.E. Warner are with the Boeing Company, PO Box 3707, MS 7L-70, Seattle, WA 98124-2207. E-mail: {david.h.jones, william.e.warner}@boeing.com.
- W. Chan was with the Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350.

Manuscript received 15 Nov. 1999; revised 8 Apr. 2000; accepted 1 May 2000. Recommended for acceptance by D. Garlan. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112146.

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 5.1)

- *Abstraction* to decrease the *number of BDD variables*. Given a property to check, we perform a simple dependency analysis to generate a reduced model that is guaranteed to give the same results as with the full model (Section 5.2).
- *Short-circuiting* to reduce the *number of BDDs* generated by stopping the iterations before a fixed point is reached (Section 5.3).

We provide experimental results showing how each of our techniques affected the performance of the analyses. Techniques like short-circuiting and abstraction are conceptually straightforward and applicable to many systems. Most other techniques were designed to exploit the semantics and environmental assumptions of many statecharts models. More specifically, our model of statecharts responds to environment inputs by performing a macrostep, divided into a number of microsteps synchronized by events. Many of our techniques take advantage of the *synchrony hypothesis*, which says that a macrostep is assumed to be atomic with respect to the environment. Particularly worth mentioning is the technique of using a microstep counter pointed out above. 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.

Other contributions of this line of work are the case studies themselves. Our analysis of the TCAS II model has been described in detail elsewhere [3]. In this article, we report on the Boeing EPD case study. Formal models have been used increasingly in Boeing to specify and validate functional requirements of airborne computing systems [6]. One of the modeling languages used is statecharts, thanks to their intuitive notations, ability to scale, and the availability of supported tools [7]. Developed for research purposes, the statecharts studied in this work model a fault-tolerant electrical power distribution system designed for use on aircraft. Its purpose is to distribute electrical power from power sources to power buses via a number of circuit breakers while tolerating failures in the power sources and circuit breakers. We were reasonably confident in the correctness of the model based on simulation results, but the model-checking analysis disclosed subtle modeling and logical flaws. Our efforts have been directed at finding bugs instead of verifying correctness. We give examples to argue for early use of model checking as a debugging tool because of the lower costs for analysis and the tendency of similar errors to recur in various parts of the system.

The rest of the article is organized as follows: We give an overview of statecharts and our verification approach in Section 2. Section 3 is devoted to a brief summary of the TCAS II case study and a more detailed report on the EPD case study. Discussions on the optimization techniques are divided into two sections: Section 4 focuses on the techniques that aim at reducing the size of the BDDs representing state sets, while Section 5 describes other techniques. Related work is discussed in Section 6. We conclude in Section 7 with some overall recommendations. Preliminary results appeared elsewhere [8], [9].

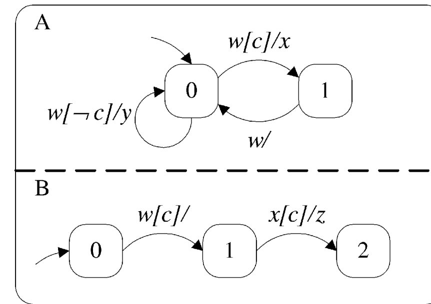

Fig. 1. A statecharts example.

2 Background

In this section, we review the syntax and semantics of statecharts and RSML and explain our approach to analyzing them using symbolic model checking.

2.1 Statecharts and RSML

The statecharts language is a popular, informally defined visual language for specifying complex reactive systems [5] and the STATEMATE toolset implements a particular semantics of statecharts [10]. RSML is another language based on statecharts with slightly different syntax and semantics [4]. We use statecharts to refer to either language when the distinctions are immaterial to this work. They both extend state-machine diagrams with parallelism, superstates, and broadcast communications. For simplicity, we discuss only a subset of their features and, in particular, will not discuss superstates in this paper; the techniques to be developed apply equally well to systems with superstates. Instead, our system model consists of a finite number of parallel local state machines with a finite set of events and inputs interacting with a nondeterministic environment.

Fig. 1 gives a simple example with two parallel state machines A and B. State machines are synchronized using events. Arrows without sources indicate the initial local states. Other arrows represent local transitions, which are labeled with the form $trig[cond]/acts$, where $trig$ is a trigger event, $cond$ is an optional guarding condition, and $acts$ is a (possibly empty) list of action events. The guarding condition is simply a predicate on local states of other state machines and/or inputs to the system; for example, a guarding condition may say that machine B is in state 0 and an input *altitude* is at least 1,000 meters. The general idea is that if the trigger event occurs and the guarding condition either is absent or evaluates to true, then the transition is enabled.

Initially, some external events, along with some inputs from the environment, arrive, marking the beginning of a macrostep. The events may enable some transitions as described above. (An example of an external event might be "an intruder entered the airspace"; an example of an input from the environment might be a specific reading from an altimeter.) A *maximal* set of enabled transitions, collectively called a microstep, is taken—the system leaves the source

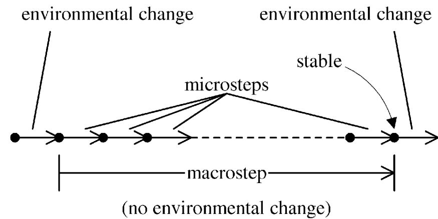

Fig. 2. Microstep, macrostep, and synchrony hypothesis.

local states, enters the destination local states, and generates the action events (if any). The system is nondeterministic if this maximal set of enabled transitions is not unique, that is, some machine can have more than one transition enabled. Otherwise, the system is deterministic. All events are broadcast to the entire system, so the generated events may enable more transitions. Unless they are regenerated by other transitions, the events disappear after one microstep. The macrostep is finished when no transitions are enabled. The semantics of RSML adopt the synchrony hypothesis from synchronous programming languages [11]: During a macrostep, the values of the inputs do not change and no new external events may arrive; in other words, the system is assumed to be infinitely faster than the environment. Fig. 2 depicts these notions. While RSML enforces the synchrony hypothesis, STATEMATE optionally allows it. We assume the synchrony hypothesis, which is central to the issues and techniques discussed in this paper.

In Fig. 1, assume that w is the only external event, c is a Boolean input, and machines A and B are in their respective state 0. When w arrives, if the input c is false, then the event y is generated. The macrostep is finished since no new transitions are enabled. Instead, if c is true when w arrives, the transitions from 0 to 1 in machines A and B are simultaneously taken and event x is generated, completing one microstep. Then, a second microstep starts. Notice that, because of the synchrony hypothesis, the input c must be true as before and the external event w cannot occur. So, only the transition from 1 to 2 in machine B is enabled and taken, generating event z and finishing the macrostep.

In RSML, the guarding condition is often very complex, so it is specified separately from the diagram in a tabular form called an AND/OR table. The guarding condition can refer to the local state of a machine m at the end of the previous macrostep, using the notation $Prev(m)$.

STATEMATE allows local transitions not guarded by events. That is, a transition can have labels of the form $[cond]/acts$. Such transitions are enabled when the system is in the source local state and *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. For simplicity, unless explicitly stated otherwise, we assume the absence of condition-driven transitions.

2.2 Encoding as a Transition System

To analyze statecharts using state-exploration techniques, we view the statecharts as a transition system (Q, R, I), where Q is a finite set of (global) states, $R \subseteq Q \times Q$ a total transition relation, and $I \subseteq Q$ a set of initial (global) states. A state in Q is a tuple of the current local state of each state machine, the set of events occurring, and the values of the environmental inputs. If (q, q') is in R, then q is a predecessor of q', and q' is a successor of q. A path is an infinite sequence of states in which each consecutive pair of states belongs in R and a trace is a path that starts with some initial state in I. A state is reachable if it appears on some trace.

We symbolically encode the global state space Q of a statecharts system by declaring a set X of state variables as follows: For each state machine m, declare a state variable whose range is the local states of m. For each event e, declare a Boolean state variable. The idea is that the variable is true if and only if the event occurs. For each input from the environment, declare a state variable with the same range (assumed finite). Clearly, this mapping from Q to the valuations of the state variables is one-to-one. We will not distinguish between a state variable and its encoded statecharts entity (local state, event, or input) because of their simple correspondence.

Given this encoding, the set of initial states I is represented as

$$I \equiv \bigwedge_{m \in M} m = m_0 \wedge \bigwedge_{e \in E_i} \neg e, \tag{1}$$

where M is the set of state machines, m_0 is the initial local state of m, and E_i is the set of internal events. This simply says that, initially, each machine is in its initial local state and all the internal events do not occur, but the external events and inputs are not constrained.

More interesting is the encoding of the transition relation R. To illustrate the idea of the encoding, we assume the system is deterministic for simplicity; for nondeterministic systems, we refer the reader to [3]. (The techniques in Sections 4 and 5 are applicable to both deterministic and nondeterministic systems.) For each state variable $var \in X$, declare a variable var' that has the same range as var and intuitively represents its next-state value. Let X' be the set of all these primed variables. We would like to define an expression over $X \cup X'$ to specify the relation R.

For each local transition t, let $src(t)$, $dst(t)$, $trig(t)$, $cond(t)$, and $acts(t)$ denote its source local state, destination local state, trigger event, guarding condition, and the set of action events, respectively. The expression $cond(t)$ is defined to be $true$ if transition t does not have a guarding condition. Define $mach(t)$ to be the current state of the machine in which t is located and an expression $en(t)$ as

$$en(t) \equiv trig(t) \wedge mach(t) = src(t) \wedge cond(t), \tag{2}$$

which represents whether transition t is enabled: It is enabled when its trigger event occurs, the machine is in the source state, and the guarding condition is true. For each machine m, define $micro_m$ as

$$\begin{aligned} micro_m \equiv & \left(\bigwedge_{t|mach(t)=m}(en(t) \rightarrow mach(t)' = dst(t))\right) \\ & \wedge \left((\bigwedge_{t|mach(t)=m} \neg\, en(t)) \rightarrow mach(t)' = mach(t)\right), \end{aligned} \tag{3}$$

which describes how the machine moves in a microstep. The first conjunct directs the machine to the destination state of an enabled transition, while the second conjunct

prohibits the machine from making any state change if none of the transitions are enabled. For each event e, define $micro_e$ as

$$micro_e \equiv \left(\bigvee_{t|e\in acts(t)} en(t)\right) \leftrightarrow e', \tag{4}$$

which describes whether an event is generated by a microstep. For each input h, define $micro_h$ as

$$micro_h \equiv \neg\, stable \rightarrow h' = h, \tag{5}$$

where $stable$ indicates whether the system is stable. We define it as

$$stable \equiv \bigwedge_{e\in E} \neg\, e, \tag{6}$$

where E is the set of all events. Now, we can define $micro$ as

$$micro \equiv \bigwedge_{m\in M} micro_m \wedge \bigwedge_{e\in E} micro_e \wedge \bigwedge_{h\in H} micro_h, \tag{7}$$

where H is the set of all inputs. Intuitively, the expression encodes all the microsteps. To encode the environmental change across macrosteps, define

$$env \equiv stable \rightarrow \left(\bigwedge_{m\in M} m' = m \wedge \bigwedge_{e\in E_i} \neg\, e'\right). \tag{8}$$

When the system is $stable$, arbitrary external events and inputs may be accepted. Finally, the transition relation R is encoded as

$$R \equiv micro \wedge env. \tag{9}$$

For RSML machines in which $Prev(m)$ for some machine m appears in some guarding condition, we declare an additional state variable p_m with the same range as m, conjoin with (1) the expression $p_m = m_0$, and conjoin (9) with the expression

$$(stable \rightarrow p'_m = m) \wedge (\neg\, stable \rightarrow p'_m = p_m). \tag{10}$$

2.3 Symbolic Model Checking

Many properties of a transition system can be expressed in the Computation Tree Logic (CTL) [12]. Its formulas are built from propositions (predicates over the state variables), the usual Boolean operators, path quantifiers **A** (for all paths) and **E** (for some path), and modalities **X** (next-time), **G** (always), and **W** (weak until), among others, with every modality immediately preceded by a path quantifier. Each modality is evaluated over a path and, intuitively, $\mathbf{X}\phi$ means that ϕ holds on the path starting at the next state, $\mathbf{G}\phi$ means that ϕ holds everywhere on the path, $\phi\mathbf{W}\psi$ means that ϕ holds everywhere before ψ holds, and ϕ must hold forever if ψ never holds. For example, the formula $\mathbf{AG}\neg error$ asserts that none of the error states are reachable and $\mathbf{AG}(request \rightarrow \mathbf{A}(request\ \mathbf{W}\ response))$ asserts that, once issued, a request will persist unless a response is given.

Given a transition system and a temporal-logic formula, the model-checking problem asks whether the transition system satisfies the formula. If not, to provide valuable diagnostic information, a model checker usually gives a *counterexample*, a trace that falsifies the property.

The truth value of a formula can be found by searching the state space. We define $Pre : 2^Q \rightarrow 2^Q$ to be the set of predecessors (also called the preimage) of any set S of states under the transition relation R:

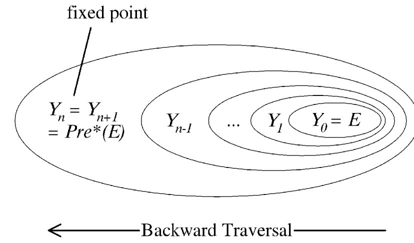

Fig. 3. Algorithm for computing $Pre^*(E)$.

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

Consider the CTL formula AG$\neg$*error* the simplest kind of temporal property and one that seems to be of common use in practice. We can characterize the model-checking problem for the formula in a set-theoretic manner: Determine whether $I \cap Pre^*(E)$ is empty, where E is the set of error states (states in which the proposition $error$ holds) and $Pre^*(E)$ is the set of states that may eventually reach an error state. More specifically, it is the least fixed point of $E \cup Pre(Y)$ or the smallest state set Y that satisfies of the equation $Y = E \cup Pre(Y)$. The existence of the fixed point is guaranteed by the finiteness of Q and the monotonicity of Pre. Fig. 3 shows an iterative algorithm for computing this fixed point. It starts with $Y_0 = E$ and iteratively computes $Y_{i+1} = Pre(Y_i) \cup Y_i$ until reaching a fixed point. The set Y_i is the set of states that may reach an error state in at most i transitions. Note that this is essentially a backward breadth-first search. All other CTL formulas can be computed similarly as (possibly multiple) fixed points [12]. In so-called "explicit" model checking, a state set is represented simply by labeling individual states in the transition system. 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 [1].

The main idea is to symbolically encode a state set as a predicate over the state variables in X, in the same way we encoded the initial global states I in (1). We can then manipulate this predicate directly to explore the whole set without enumerating its elements. Because we are dealing with finite state spaces, we can make each state variable in X Boolean by introducing (finitely many) extra state variables. Therefore, each state set S can be encoded as a Boolean function $S(X)$. The transition relation R can be similarly encoded as a Boolean function $R(X, X')$. Intersection, union, and complementation on sets, respectively, become conjunction, disjunction, and complementation on Boolean functions. Predecessor computation can now be expressed as Boolean operations as well:

$$Pre(S) = \exists X'.\ R(X, X') \wedge S(X'). \tag{11}$$

These Boolean functions can be represented as reduced ordered binary decision diagrams (BDDs) [2]. Boolean operations, satisfiability checking, and existential quantification can be performed efficiently using BDDs, which,

therefore, can be used to implement the searches described above. BDDs are canonical, meaning that each Boolean function has a unique BDD representation up to a chosen variable order.

The size of the BDDs is a major bottleneck in BDD-based algorithms. In the worst case, it can be exponential in the number of variables. In practice, though, it is often small, even when the set represented is large, but this depends on the chosen variable order and the dependencies among the variables.

3 Case Studies

We carried out two case studies to investigate the effectiveness of symbolic model checking for verifying software models specified in statecharts or RSML. This section briefly reviews our previous case study on an airborne collision avoidance system and describes in more detail another case study on an aircraft electrical power distribution system. In both experiments, despite prior verification efforts using other techniques, we discovered violations of nontrivial properties using model checking. (However, many of the analyses were intractable without the optimizations explained in Sections 4 and 5.) We used and modified CMU's BDD-based model checker SMV [13] release 2.4.4 in our studies.

3.1 TCAS II

In a previous case study, we analyzed a portion of a preliminary version of the system requirements specification of the Traffic Alert and Collision Avoidance System II (TCAS II). The 400-page document is written in RSML. TCAS II is required by the US Federal Aviation Administration on most commercial aircraft that enter US airspace. When another aircraft intrudes into a defined volume surrounding the TCAS-equipped aircraft, TCAS II generates warnings and suggests possible escape maneuvers, called resolution advisories (RAs), to the pilot. These RAs include "Climb," "Descend," "Do not climb more than 1,000 feet per minute," etc.

The complexity of the system stems from the vast number of inputs from the pilot, altimeter readings, and ground stations, the complicated logic for deriving RAs to maintain a safe separation while minimizing disruption, the needs for avoiding false alarms, etc. This complexity is partly reflected in the RSML specification as complicated guarding conditions, some of which occupy many pages of description. They contain predicates of local states and of the input variables and often involve nontrivial arithmetic predicates. While many other researchers conservatively abstract each arithmetic predicate as an independent Boolean variable [14], [15], [16], we encode each bit of the numeric inputs as a Boolean variable, resulting in more accurate analysis at the expense of requiring more Boolean variables. In addition, a guarding condition can refer to any part of the system, so the interdependencies between the BDD variables are high. These all imply relatively large BDDs for guarding conditions. On the plus side, the control flow of the state machines is simple by design and concurrency among the state machines is minimal. As we will see, some of the techniques presented later attempt to exploit these simple synchronization patterns.

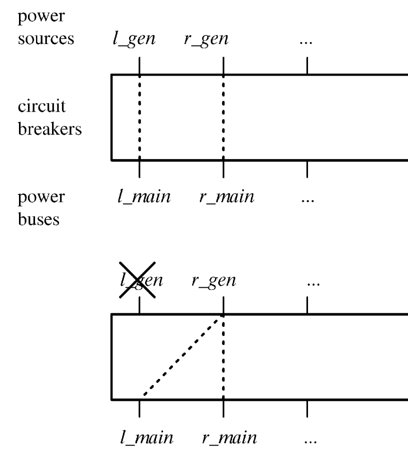

Fig. 4. Handling a power-source failure in the EPD model. The top is (a) and the bottom is (b).

We encoded the global state space of the portion of the specification with 227 Boolean variables, 10 of which are for events, 55 for the local states, 134 for altitude and altitude rates (which are integers), 22 for inputs other than altitude and altitude rates, and six for other purposes. The size of the state space is about 1.4×10^{65}. The size of the *reachable* state space is at least 9.6×10^{56}. We checked that certain transitions are mutually exclusive (otherwise, it would indicate inconsistencies in the specification), a descent RA should not be given when the aircraft is close to the ground, two of the outputs should agree with each other, etc. Several anomalies were discovered using model checking. The reader is referred to [3] for details.

3.2 EPD System

In our second case study, we analyzed a statecharts model of the electrical power distribution (EPD) system on the Boeing 777 aircraft. Several faults of the model were uncovered, although we were quite confident about its correctness based on simulation results. We briefly describe the model, discuss some results of the analysis, and suggest how model checking could potentially be used to benefit the model-based development processes used at Boeing.

3.2.1 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.

Fig. 4a depicts part of the system configuration in normal operations. The power buses l_main and r_main belong to the main AC power subsystem and are normally powered by the generators l_gen and r_gen, respectively. When l_gen loses its power because of either manual shutdown or failure, the circuit breakers will be reconfigured automatically to use r_gen to power both l_main and r_main, as illustrated in Fig. 4b. The same configuration may also result from failures in the circuit breakers that connect l_gen

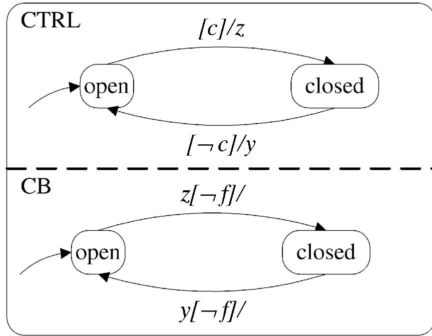

Fig. 5. A circuit breaker (*CB*) and its controller (*CTRL*).

and l_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. Fig. 5 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 to the local states of other circuit breakers and the power sources. Compared with the TCAS II specification, though, these guarding conditions seem less complicated and do not involve arithmetic. However, the model's synchronization structure is not as simple as that of TCAS II.

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 [7]. 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.

3.2.2 Results of Analyses

The analyses can be divided into analyses on normal behaviors (i.e., no component failures) and fault tolerance (single and multiple failures). Here, we report some of the more interesting results. 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

$$\mathbf{AG}((stable \wedge no\text{-}failures) \rightarrow (main \wedge backup)), \qquad (12)$$

where $no\text{-}failures$ is a proposition indicating the absence of failures (each of the 17 failures is represented by an atomic proposition), and $main$ and $backup$, respectively, assert 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

$$\mathbf{AG}((stable \wedge no\text{-}failures) \rightarrow seperate\text{-}sources), \qquad (13)$$

where the proposition $seperate\text{-}sources$ 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 Fig. 5. 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

$$\mathbf{AG}((stable \wedge at\text{-}most\text{-}1\text{-}failure) \rightarrow main), \qquad (14)$$

where the proposition $at\text{-}most\text{-}1\text{-}failure$ 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 to 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 actually realized that the property 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:

$$\mathbf{AG}((stable \wedge at\text{-}most\text{-}1\text{-}failures) \rightarrow at\text{-}least\text{-}1\text{-}backup). \qquad (15)$$

The model checker responded with a counterexample exposing a logical flaw similar to B2 above. The counter-example 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

$$\mathbf{AG}(main \rightarrow \mathbf{A}(main\mathbf{W}\neg no\text{-}failure)), \qquad (16)$$

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.

3.2.3 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 [6]. 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.

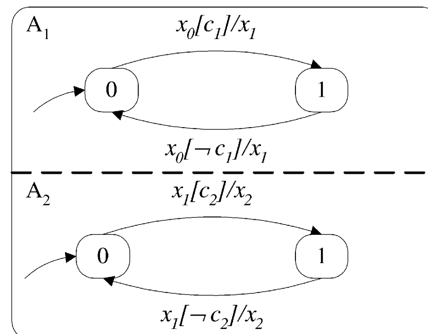

Fig. 6. State machine with mutually exclusive events and nonoblivious synchronization.

4 State-Set Optimizations

Many of the analyses in the case studies were feasible only after we performed various optimizations. A major bottleneck in the analyses was the size of the BDDs representing state sets. In this section, we will see several techniques that we used to reduce the BDD size, sometimes by orders of magnitude. Some of the techniques are quite different from the common techniques used in hardware verification and are perhaps even counterintuitive. 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. Note that the TCAS II model analyzed was slightly different from the one examined in the initial study [3], so the data reported there should not be directly compared with the results in this paper.

4.1 Pruning Backward Traversals

Recall from Section 2.3 that, to evaluate the CTL formula $\mathbf{AG}\neg error$, we compute the fixed-point $Pre^*(E)$ 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.

4.1.1 An Example with Mutually Exclusive Events

Consider the system in Fig. 6. 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. Note that at most one local transition can be enabled at any time. In particular, the transitions in A_2 can only be taken

TABLE 1
Performance of Pruning for TCAS II

	T1		T2		T3		T4		T5	
	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)
Base	79	400	182	713	257	1060	342	1090	∞	
MX	11	110	20	123	76	369	38	152	47	249
Ratio	7.2	3.6	9.1	5.8	3.4	2.9	9.0	7.2	—	

Time is in seconds and number of BDD nodes is in thousands. MX represents pruning using mutually exclusive events and ∞ indicates timeout after one hour.

after those in A_1. (The example also demonstrates "nonoblivious synchronization," which we will discuss later in Section 4.2.).

However, a backward traversal may consider many simultaneous transitions, which cannot occur in any execution. More explicitly, suppose we want to check whether machines A and B can both be in state 1 simultaneously. Traversing backward, we find that, one microstep before, the system may be in one of the three situations: A in 0 and B in 1, A in 1 and B in 0, or both machines in 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.

4.1.2 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 Σ is the set of mutually exclusive events, we can compute the conjunction of R and

$$\bigwedge_{\substack{e_1,e_2\in\Sigma\\ e_1\neq e_2,}} \neg(e_1 \wedge e_2),$$

and use the result as the transition relation in the traversals.[1]

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. We define $\prec$ to be a 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 $e_1[c]/e_2$ for some guarding condition c. We assume that $\prec$ is acyclic, that is, $(e,e) \notin \prec^+$ for each e, where $\prec^+$ is the (nonreflexive) 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 ith 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 Fig. 6, 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.

4.1.3 Experimental Results

Table 1 shows the results of applying the technique to the TCAS II model. Properties T1 through T4 refer to Increase-Descent Inhibition, Function Consistency, Transition Consistency, and Output Agreement described in [3]. Property T5 refers to an assertion in [17, p. 49] that two machines, called *Corrective-Climb* and *Corrective-Descend*, should not be in their local states *Yes* simultaneously (comments in our version of the TCAS II requirements, however, explicitly say that the two local states are not mutually exclusive). The property was proven false by the model checker. This property was infeasible to check without pruning. For other properties, the speedup obtained was as much as a factor of 9.

4.2 Oblivious vs. Nonoblivious 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 4.3. 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.

Fig. 7 shows a system whose stable-state behaviors are identical to the one in Fig. 6. The main difference between the two systems is that, in Fig. 6, event x_1 always occurs after x_0, while, in Fig. 7, event x_1 is not generated if there is no state change in machine A_1. In other words, in Fig. 7, an event signals the *completion* of a state machine's execution

1. 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.

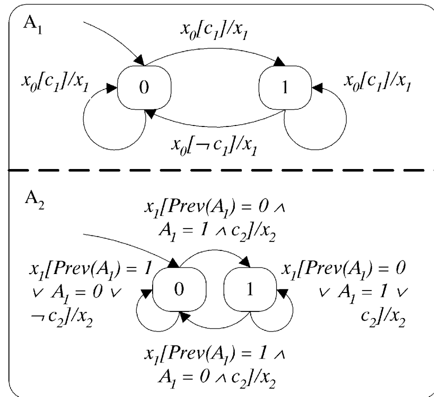

Fig. 7. Oblivious Synchronization The stable-state behaviors are identical to those in Fig. 6.

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 Fig. 6, an event signals a state *change* and the sequence of events generated depends on which local transition is taken; we call such systems nonoblivious. 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 nonoblivious 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.[2] 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 [10, p. 323]. Notice, however, that the differences arise not from the semantics of the language, 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 Fig. 7.

4.2.1 Difference in Efficiency of Analyses

Whether one style is better than the other for specification purposes is beyond the scope of this article. 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 nonoblivious 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.

2. Indeed, the simple control flow prompted [18] to get rid of events altogether in their new requirements language SpecTRM-RL.

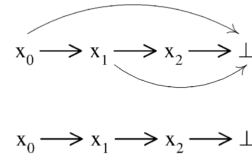

Fig. 8. The extended precedence relation $\prec$. The top is (a), nonoblivious; the bottom is (b), oblivious.

Intuitively, the decoupled synchronization and local control of oblivious systems induces fewer dependencies among the state variables, potentially keeping the BDDs smaller. In particular, nonoblivious 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 nonoblivious 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 4.1.2) to a binary relation over the events E together with a special symbol $\perp$, which intuitively represents stable states: In addition to the definition given earlier, for each event e, we have $e \prec \perp$ if there exists a global state q in which e occurs and q has a stable successor state. Fig. 8 shows the extended precedence relations for the nonoblivious and oblivious systems in Figs. 6 and 7. Note that there are fewer edges pointing to $\perp$ in Fig. 8b; for example, the edge $(x_0, \perp)$ is absent because x_0 always triggers x_1 in Fig. 8a 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. Fig. 9 shows the intuition for the nonoblivious model. In Fig. 9a, we lay down every possible sequence of events in a macrostep in the nonoblivious system; the diagram is obtained from Fig. 8a by tracing all the paths starting from the external event x_0. The backward search can now be illustrated by traversing backward from $\perp$ in a breadth-first manner. Figs. 9b, Fig. 9c, and Fig. 9d show 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 Fig. 9c, in the next iteration in Fig. 9d, 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.

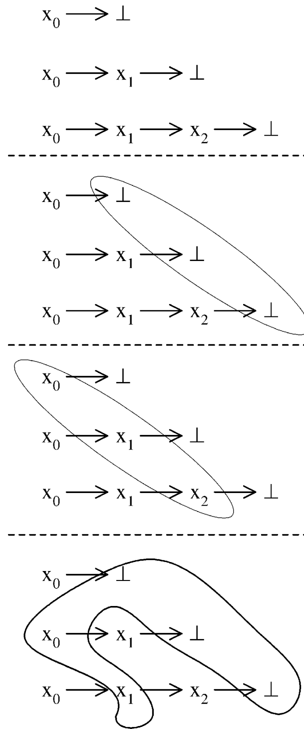

Fig. 9. Backward search for the nonoblivious model. The topmost is (a), representing all event sequences. The next three are (b), (c), and (d), representing the first, second, and third iterations, respectively.

Fig. 10 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, 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.

4.2.2 Experimental Results

As mentioned, although the analyses of the oblivious TCAS II model were generally successful, our initial attempt to analyze the nonoblivious 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

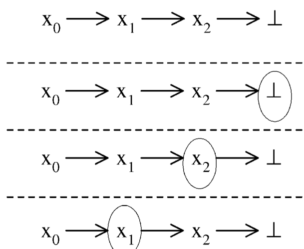

Fig. 10. Backward search for the oblivious model. The top is (a), showing the only possible event sequence. The next three are (b), (c), and (d), showing the first, second, and third iterations, respectively.

TABLE 2
Performance (Time in Seconds) for Computing Fixed Points for the Parameterized Examples

n	Non-Oblivious			Oblivious		
	base	MX	MC	base	MX	MC
5	0	0	0	0	0	0
10	4	2	1	1	1	1
15	55	33	3	7	5	4
20	358	133	7	27	19	12

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 Figs. 6 and 7 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. The Appendix lists the SMV programs used.

Table 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 4.3.) The results show that the nonoblivious 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 nonoblivious 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 nonoblivious models more efficient.

4.3 Microstep Counter

In Sections 4.1 and 4.2, 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.

4.3.1 Construction of the Microstep Counter

In Section 4.1, 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 ith microstep. Observe that the maximum length l of a macrostep is the largest integer in $\sigma(e)$ for any e. For Fig. 6, we have $\sigma(x_0) = \{1\}$, $\sigma(x_1) = \{2\}$, $\sigma(x_2) = \{3\}$, and, therefore, $l = 3$. Note that some macrosteps may have fewer than l microsteps.

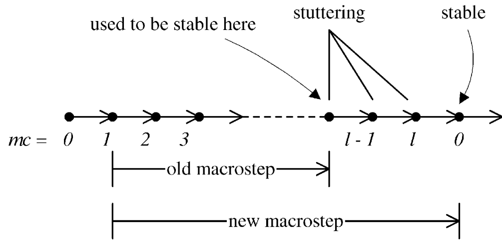

Fig. 11. Effects of microstep counter on system behaviors.

Now, to symbolically encode a statecharts model as a transition system, in addition to the usual state variables defined in Section 2.2, we declare a microstep counter mc to range from 0 to l. The behavior of the microstep counter depends only on the set E_x of external events.

Modification 1 (Microsteop counter). Let s denote $\bigvee_{e\in E_x} e$ (some external event occurs in the current state) and s' denote $\bigvee_{e\in E_x} e'$ (some external event occurs in the next state). We conjoin the symbolic encoding of the initial states I in (1) with

$$(\neg s \rightarrow mc = 0) \wedge (s \rightarrow mc = 1)$$

and conjoin the transition relation R in (9) with

$$\begin{aligned} &((mc = 0 \wedge \neg s') \rightarrow mc' = 0) \\ \wedge\, &((mc = 0 \wedge s') \rightarrow mc' = 1) \\ \wedge\, &(0 < mc < l \rightarrow mc' = mc + 1) \\ \wedge\, &(mc = l \rightarrow mc' = 0). \end{aligned}$$

Stability now depends only on the microstep counter:

Modification 2 (Stability). The expression $stable$ is now defined as $mc = 0$.

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 l. 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 l microsteps.

Clearly, the local transitions in the statecharts are unaffected by the changes, but the stable states may be delayed, as illustrated in Fig. 11—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 *stutters* in the interim [19] and all "stutter-invariant" CTL formulas, including those without the next-time **X** operator, are preserved by stuttering [20]. (Formulas with the **X** operator can count the number of microsteps and thus may not be preserved.)

Our final modification uses the microstep counter to guard transitions.

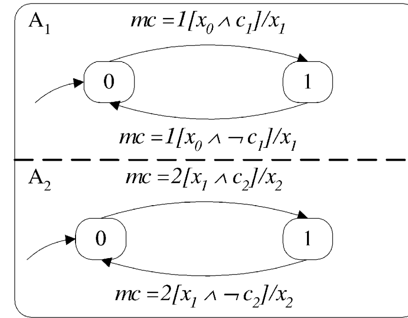

Fig. 12. Transitions guarded with microstep counter.

Modification 3 (Guards). For each transition t, the expression $en(t)$ in (2) is now defined as

$$mc \in \sigma(trig(t)) \wedge trig(t) \wedge mach(t) = src(t) \wedge cond(t).$$

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

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

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 as decoupling 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 proven by induction on the definition of σ. So, the inclusion of $mc \in \sigma(e_1)$ is redundant as far as forward behaviors are concerned. We make the following claim:

Claim 1 Correctness. *If the relation $\prec$ is acyclic, then Modifications 1-3 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).*

4.3.2 Benefits of Microstep Counter

To see how these modifications help, consider again our nonoblivious system in Fig. 7. Fig. 12 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 Fig. 8b instead of Fig. 8a. Fig. 13 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.

To see that the modifications help prune unreachable simultaneous transitions in backward searches, observe that the microstep counter in Fig. 12 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

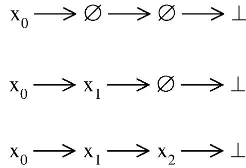

Fig. 13. Event sequences of the nonoblivious system with microstep counter. $\emptyset$ indicates a stuttering state in which no events occur.

exclusion of events in Section 4.1. 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.

4.3.3 Experimental Results

The use of microstep counters was crucial for the analyses of the EPD model discussed in Section 3.2—none of the interesting properties could be analyzed within two hours of CPU time without using the optimization. Table 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 2 show the results of applying the technique to the parameterized examples. (The Appendix gives the SMV code used in the experiment.) The microstep counter dramatically improved the performance for the nonoblivious models by up to a factor of 19, making them more efficient to analyze than the oblivious models. Here, the slight advantage of the nonoblivious models stems from the fewer number of state variables because the previous states of the machines are not encoded.

4.3.4 Condition-Driven Transitions

Extending our techniques to handle condition-driven transitions requires a more general framework and we omit the details here. The basic idea, though, remains the same. Specifically, when we encode a transition t, we want to conjoin its guarding condition with a new proposition $mc \in \rho(t)$, where $\rho(t)$ is a set that includes an integer i if transition t can be taken in the ith microstep. For systems with every transition triggered by some event, the set $\rho(t)$ is simply $\sigma(e)$ with e being the trigger event of t, as we saw in Modification 3. In the presence of condition-driven transitions, we can still compute ρ statically in many common cases, although the procedures are more involved.

4.4 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

TABLE 3
Performance of Using Microstep Counter for the EPD Model

	E1	E2	E3	E4	E5	forward
time (sec.)	49	54	85	89	462	3806
node (K)	464	402	837	452	2965	3865

Without the microstep counter, every formula could not be evaluated within two hours of CPU time. Properties E1 through E5 refer to (14), (13), (12), (16), and (15) in Section 3.2.2, respectively. The order refers to the relative difficulty (based on time) in analyzing them. The rightmost column will be explained in Section 4.4.

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.

4.4.1 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 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 [21], [22].

Partly inspired by [23], 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 4.1, invariants can sometimes improve search efficiency, paradoxically, keeping all invariants can hurt performance. Consider again machine A_1 in Fig. 7. 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, the BDD for the invariant is as well. There are likely to be many such implicit invariants in the system and their conjunction may have a large BDD representation even if

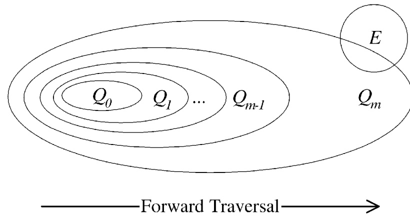

Fig. 14. Standard forward traversal underlying original counterexample search algorithm.

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, 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. However, this is not the case in our statecharts models.

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).

4.4.2 Implications on Counterexample Search

In addition to fixed-point computations, the 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, the model checker that we used, uses a forward search to find counterexamples and suffers from the BDD blowup pointed out above. Fig. 14 shows the original counterexample search algorithm used in the model checker. Let Q_0 be any nonempty subset of $Pre^*(E) \cap I$. Iteratively compute $Q_{i+1} = Succ(Q_i) \cup Q_i$ until reaching E. 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 (Fig. 15) by tracing backward from $Q_m \cap E$. 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 $q_0, q_1, \ldots, q_m$.

The first, forward traversal in Fig. 14 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 Fig. 16, is to remember every Y_i computed in Fig. 3 (our actual implementation stores the difference $Y_i \setminus Y_{i-1}$ instead of Y_i) and use them to restrict the counterexample search. Start with some $q_0 \in Y_n \cap I$ and iteratively pick some $q_i \in Succ(q_{i-1}) \cap Y_{n-i}$ to obtain a counterexample $q_0, q_1, \ldots, q_n$.

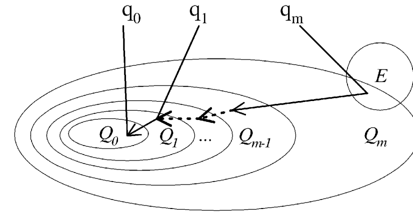

Fig. 15. Original Algorithm for counterexample search.

This same algorithm is used in other model checkers as well [23]. 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.

5 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 section.

5.1 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 [24]. 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.

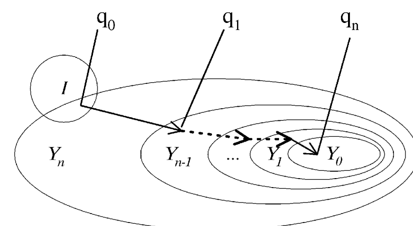

Fig. 16. Modified algorithm for counterexample search. The sets Y_i are computed in Fig. 3.

5.1.1 Background

We first review the idea of conjunctive and disjunctive partitioning. The transition relation R is sometimes given as a disjunction $D_1 \vee D_2 \vee \cdots \vee D_j$ 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 computation shown in (11) in Section 2.3 can be easily modified by distributing the existential quantification over the disjunction. We thus have

$$\begin{aligned} Pre(S) &= \exists X'.\, R(X,X') \wedge S(X') \\ &= \exists X'.\, (D_1(X,X') \vee D_2(X,X') \vee \cdots \\ &\quad \vee D_j(X,X')) \wedge S(X') \\ &= d_1(X) \vee d_2(X) \vee \cdots \vee d_j(X), \end{aligned}$$

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

$$d_i(X) = \exists X'.D_i(X,X') \wedge S(X').$$

So, we can compute the predecessors without ever building the BDD for R. Successor computation is symmetric.

If R is given as a conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_k$, 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

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

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 $C_1, C_2, \ldots, C_k$.

5.1.2 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: 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 [25] and [26] 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.

5.1.3 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:

$$\begin{aligned} R &= (\alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_j) \wedge R \\ &= D_1 \vee D_2 \vee \cdots \vee D_j, \end{aligned}$$

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

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

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.

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

for $1 \leq i < j$, and

$$\begin{aligned} \alpha_j &= \neg u_1 \wedge \neg u_2 \wedge \cdots \wedge \neg u_{j-1} \\ \alpha_{j+1} &= \neg \alpha_1 \wedge \neg \alpha_2 \wedge \cdots \wedge \neg \alpha_j. \end{aligned}$$

In other words, α_i corresponds to the states in which only u_i has just occurred, α_j, none of the events have, and α_{j+1}, at least two of the events have. They clearly form a cover. We made two observations. First, we can drop α_{j+1}, which is a contradiction because of the mutual exclusion assumption. Second, the conjuncts in our symbolic representation of R in Section 2.2 have antecedents which are predicates over the events. For example, $en(t)$ requires its trigger event $trig(t)$ to be true. If the event is, say, u_i for some $1 \leq i < j$, then the BDD for $trig(t)$ is relevant only to the disjunct D_i. 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 4.1.2.

5.1.4 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. Fig. 17 shows an example in which an event x triggers two state machines. If

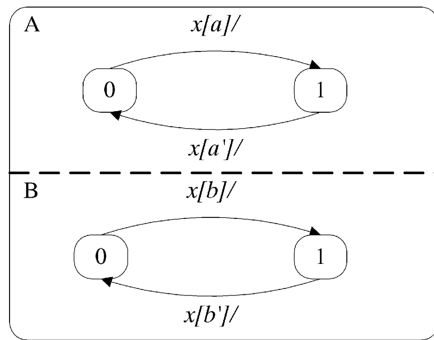

Fig. 17. One event triggering two state machines.

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.

Alternatively, we may serialize the complicated microstep into cascading microsteps to reduce the BDD size. Fig. 18 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.

5.1.5 Experimental Results

TCAS II. Table 4 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 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

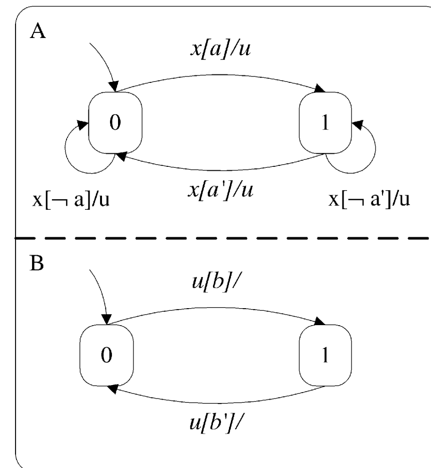

Fig. 18. The serialized machine.

in SMV.[3] 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. This mistake—made early in the previous study and discovered quickly by inspection—omitted some self-loops similar to those in Fig. 7, which effectively made the system nonoblivious and produced 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 disproportionally 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.

3. Actually, we implemented a simple improvement that was used in all results, including this base analysis. Recall that a 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 [24]. SMV performs this optimization except when conjunctive partitioning is used. We simply changed SMV to eliminate this limitation.

TABLE 4
Performance of Different Ways of Partitioning TCAS II

				Building BDDs for R		T1		T2		T3		T4		T5		
	Full Model															
	No. of fixpoint iterations					24		29		29		38		26		
	Optimizations															
	MX	CP	DP	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	
1	—	—	—	20	93	79	400	182	713	257	1060	342	1090	∞		1
2	—	√	—	33	176	**40**	**273**	**97**	**345**	**147**	**488**	**193**	**412**	∞		2
3	√	—	—	20	94	11	**110**	20	**123**	76	369	38	152	47	249	3
4	√	√	—	25	166	9	170	18	190	**51**	267	31	215	39	245	4
5	√	—	√	34	464	18	464	33	464	798	968	34	463	74	480	5
6	√	√	√	40	128	**7**	128	**14**	139	57	**217**	**24**	**150**	**29**	**160**	6
	Mistranslated Model															
	No. of fixpoint iterations					24		29		29		38		26		
	Optimizations															
	MX	CP	DP	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	
7	√	—	—	20	93	285	697	317	1016	95	314	518	1129	615	2245	7
8	√	√	—	26	174	323	1043	791	1546	91	424	497	1471	∞		8
9	√	—	√	36	462	972	843	1117	964	358	895	1340	952	∞		9
10	√	√	√	42	126	**126**	**327**	**154**	**515**	**49**	**185**	**215**	**398**	**213**	**678**	10
	Serialized Model															
	No. of fixpoint iterations					36		41		45		54		38		
	Optimizations															
	MX	CP	DP	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	time (s)	node (K)	
11	√	—	—	27	103	12	**111**	39	190	**127**	311	46	**144**	89	325	11
12	√	√	—	31	167	12	167	38	234	**127**	323	44	199	94	363	12
13	√	—	√	27	139	12	139	40	**161**	136	251	**32**	160	76	**177**	13
14	√	√	√	48	136	**11**	136	**34**	162	129	**221**	39	156	**74**	196	14

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

For each group of experiments, the best time and space requirements for each property are shown in bold face. An entry with ∞ indicates timeout after one hour.

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 trade-off 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 3 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.

5.2 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.

5.2.1 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, 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.[4]

(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

4. In the presence of hierarchical states, the superstate also becomes relevant.

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 4.1.2), 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 (5). 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 Fig. 18 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 [15]. However, the syntax sometimes allows easy detection of most false dependencies of this kind—for example, the self-loops in Fig. 18 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 Fig. 19, 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.

5.2.2 Experimental Results

Table 5 shows the performance of analyzing the abstract models. The reductions obtained for our TCAS II model were significant (although one of the properties still could not be analyzed without using other optimizations). 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 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.

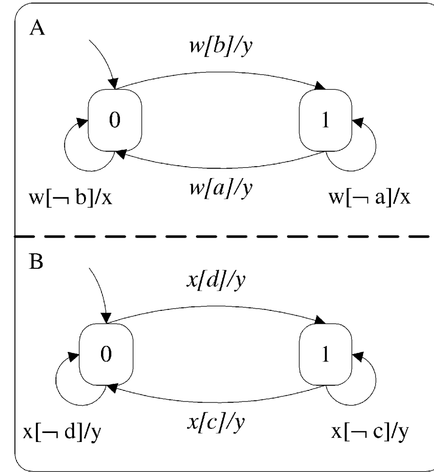

Fig. 19. False dependency: Event y does not depend on any guarding condition.

5.3 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 applied 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 6 shows the reductions obtained in our case studies.

6 Discussion and Related Work

Some of our techniques 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 [27] or asynchrony [28]. Because our models lack these special structures, these techniques are not applicable. Other hardware techniques work for general classes of circuits [29], [23], [30]. 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 [31], who propose doing an *approximate* forward traversal to compute a superset of the 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. They also independently propose disjunctive partitioning for synchronous circuits [32]. They require the designer to come up with a partition manually, while we focus on statecharts and exploit mutually exclusive events. Yang et al. [33] 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 in Sections 4.1 and 4.3 use information from forward analysis on event

TABLE 5
Performance for the Abstract Models

	time (s)	node (K)	bits	time (s)	node (K)	bits	time (s)	node (K)	bits	time (s)	node (K)	bits	time (s)	node (K)	bits
TCAS II	**T1**			**T2**			**T3**			**T4**			**T5**		
Full	79	400	227	182	713	227	257	1060	227	342	1090	227	∞	∞	227
Abstract	5	65	142	17	93	142	72	362	150	26	115	142	∞	∞	150
Ratio	15.8	6.2	1.6	10.7	7.7	1.6	3.6	2.9	1.5	13.2	9.5	1.6	—	—	1.5
EPD	**E1**			**E2**			**E3**			**E4**			**E5**		
Full	49	464	117	54	402	117	85	837	117	89	452	117	462	2965	117
Abstract	36	460	70	40	395	70	85	837	117	66	415	70	462	2965	117
Ratio	1.4	1.0	1.7	1.4	1.0	1.7	1.0	1.0	1.0	1.3	1.1	1.7	1.0	1.0	1.0

Columns marked "bits" indicate the numbers of Boolean state variables in the models. Microstep counters were used in the EPD models, but not in the TCAS II models.

TABLE 6
Performance of Short-Circuiting

	time (s)	node (K)	iter	time (s)	node (K)	iter	time (s)	node (K)	iter	time (s)	node (K)	iter	time (s)	node (K)	iter
TCAS II	**T1**			**T2**			**T3**			**T4**			**T5**		
Full	79	400	24	182	713	29	257	1060	29	342	1090	38	∞	∞	26
SC	62	400	15	143	713	15	61	669	11	136	751	24	∞	∞	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
EPD	**E1**			**E2**			**E3**			**E4**			**E5**		
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

Columns marked "iter" indicate the numbers of iterations required. SC indicates results with short-circuiting. Property E4 (16) requires two nested fixed points to evaluate and only the outer one was short-circuited. 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.

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.

Using abstraction to facilitate analysis is a very old idea [34], [35] and has also been applied to state-based software specifications. For example, [36] uses a dependency analysis technique similar to the one described Section 5.2.1, but their motivation is to facilitate manual review of the TCAS II requirements, rather than automatic verification. In the context of verifying SCR requirements, [37] suggests two automatic abstraction techniques, one of which is similar to our dependency analysis. (Of course, there are many other dependence analyses in other domains; just one example is work in program slicing [38], [39], [40].)

Note that, under our encoding in Section 2.2, a macrostep is represented as a sequence of global transitions. An alternative is to represent a macrostep as a single global transition, but this would prevent us from analyzing behaviors within a macrostep, such as (16) in Section 3.2.2. In addition, we need to symbolically compose the micro-steps to compute the macrosteps, which is not always straightforward. The efficiency of model checking is also affected: This method may blow up the BDD size for the global transition relation, but reduces the number of search iterations to reach fixed points and reduces the difference between oblivious and nonoblivious systems. Therefore, it is not clear a priori whether this method works better or worse. In our initial TCAS II experiments, it resulted in huge BDDs and poor performance and we have not considered this method further in our case studies. However, this may be a viable approach for other statecharts examples.

Some of our techniques involve transforming the model and preserve only stutter-invariant properties. Although this restriction prevents us from specifying what will happen in the next *microstep* (e.g., $\mathbf{AX}p$), we can still assert what holds in the next *macrostep* (e.g., $\mathbf{A}(\neg \mathit{stable}\ \mathbf{W}\ \mathit{stable} \wedge p)$).

We add that, in our EPD analyses, fixing the bug B1 in the model with the microstep counter dramatically reduces the time taken to evaluate each formula to less than 15 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.

7 Conclusion

We have made several contributions in this work. We carried out a case study of applying BDD-based model

checking to a statecharts specification developed at Boeing and discovered subtle flaws in the model. The combined experience from this and our previous TCAS II case study allowed us to develop some intuitions about BDD blow-ups in our domain. For example, we found that forward searches and nonoblivious synchronization tend to be less efficient for our analysis. Based on these and other intuitions, we devised techniques to optimize the performance of model checking. Although these results do not make model checking a "push the button" technology for software-oriented specifications, they take steps in that direction. Some improvements were crucial as they allowed analysis that used to be infeasible to complete in just several minutes of CPU time.

The two case studies are complementary in that the two statecharts models are written in very different ways—the TCAS II specification has a very special synchronization structure (oblivious, with most events mutually exclusive), while the EPD model is closer to other statecharts specifications. Indeed, certain techniques that worked well for TCAS II were not applicable to the EPD model (Sections 4.1 and 5.1.4). In general, we believe that the technique of using a microstep counter would facilitate the analysis of most statecharts, whereas the pruning and the DNF partitioning techniques would be appropriate for models with synchronization structures similar to that of TCAS II. The abstraction and short-circuiting techniques are general enough that they can and should always be used.

Getting intuition about 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 affect 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.

APPENDIX

This appendix gives the SMV programs for the experiments on the parameterized examples summarized in Table 2 in Section 4.2.2. They show the differences in the efficiency of analyzing oblivious and nonoblivious models, as well as the effects of pruning using mutually exclusive events and microstep counters.

The following shows the programs for the nonoblivious models in Fig. 6 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 := !x_0 & !x_1 & ... & !x_n;
VAR
  x_z : boolean;
ASSIGN
  next(x_z) := case
    stable: {0,1};
    1: 0;
    esac;
For each i with 1 ≤ i ≤ n:
DEFINE
  t_i^1 := x_{n-1} & a_i = 0 & c_i;
  t_i^0 := x_{n-1} & a_i = 1 & !c_i;
VAR
  c_i: boolean;
  a_i: boolean;
  x_i: boolean;
ASSIGN
  init(a_i) := 0;
  next(a_i) := case
    t_i^0: 0;
    t_i^1: 1;
    1: a_i;
    esac;
  next(c_i) := case
    stable: {0,1};
    1: 0;
    esac;
  init(x_i) := 0;
  next(x_i) := t_1^1 | t_1^0;
SPEC
  AG !(stable & a_{n-1} = 0 & a_n = 1)
```

Other models are obtained by modifying the code above.

- For the oblivious models in Fig. 7:
  - The definitions of t_i^1 and t_i^0 are changed to the following.

```
DEFINE
  t_1^1 := x_z & c_1;
  t_1^0 := x_z & !c_o;
For each i with 1 < i ≤ n:
  t_i^1 := x_{n-1} & ((a_i = 0 & pa_{i-1} = 0 & a_{i-1} = 1 & c_i)
               | (a_i = 1 & (pa_{i-1} = 0 | a_{i-1} = 1 | c_i)));
  t_i^0 := x_{n-1} & ((a_i = 1 & pa_{i-1} = 1 & a_{i-1} = 0 & !c_i)
               | (a_i = 0 & (pa_{i-1} = 1 | a_{i-1} = 0 | !c_i)));
```

  - The following code is added. The variable pa_i encodes the previous state of machine A_i.

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

- For pruning using mutually exclusive events (Section 4.1), the following code is added:

```
TRANS !(x_0 & x_1)
TRANS !(x_0 & x_2)
  ⋮
TRANS !(x_0 & x_n)
TRANS !(x_1 & x_2)
  ⋮
TRANS !(x_{n-1} & x_n)
```

- For microstep counters (Section 4.3):
  - The following code is added

```
VAR
 mc: 0..n;
ASSIGN
 init(mc) := x_0;
 next(mc) := case
  mc = 0: next(x_0);
  1: (mc + 1) mod (n + 1);
  esac;
```

  - The expression `stable` is redefined as

```
DEFINE stable := mc = 0;
```

  - An extra conjunct `mc = ` i is added to the definition of every t_i^0 and t_i^1.

The variable order used was (`mc`,) $x_0, c_1, a_1, (pa_1,) x_1, c_2, a_2, (pa_2,) \ldots, x_n,$ where `mc` was used only in the models with a microstep counter and the pa_is were used only in the oblivious models.

ACKNOWLEDGMENTS

The authors thank Steve Burns for observing the inefficiency of the algorithm in Fig. 14 and implementing the one in Fig. 16 in SMV and Greg Taleck for his initial work on translating and analyzing the EPD model. This work was supported in part by US National Science Foundation grants CCR-9706070 and CCR-9700660. William Chan's work was supported in part by a Microsoft-endowed graduate fellowship.

REFERENCES

[1] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation,* vol. 98, pp. 142–170, June 1992.

[2] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers,* vol. 35, no. 8, pp. 677–691, Aug. 1986.

[3] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese, "Model Checking Large Software Specifications," *IEEE Trans. Software Eng.,* vol. 24, no. 7 pp. 498–520, July 1998.

[4] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 9 pp. 684–707, Sept. 1994.

[5] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, pp. 231–274, June 1987.

[6] C.R. Nobe and M.G. Bingle, "Model-Based Development: Five Processes Used at Boeing," *Proc. IEEE Int'l Conf. and Workshop: Eng. of Computer-Based Systems,* Mar./Apr. 1998

[7] C.R. Nobe and W.E. Warner, "Lessons Learned from a Trial Application of Requirements Modeling using Statecharts," *Proc. Second Int'l Conf. Requirements Eng. (ICRE '96),* pp. 86–93, Apr. 1996.

[8] W. Chan, R.J. Anderson, P. Beame, and D. Notkin, "Improving Efficiency of Symbolic Model Checking for State-Based System Requirements," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis ISSTA '98,* M. Young, ed., pp. 102–112, Mar. 1998, Published as *Software Eng. Notes,* vol. 23, no. 2, Mar. 1998.

[9] W. Chan, R.J. Anderson, P. Beame, D.H. Jones, D. Notkin, and W.E. Warner, "Decoupling Synchronization from Logic for Efficient Symbolic Model Checking of Statecharts," *Proc. 1999 Int'l Conf. Software Eng. (ICSE '99),* pp. 142–151, May 1999.

[10] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. and Methodology,* vol. 5, pp. 293–333, Oct. 1996.

[11] G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming,* vol. 19, pp. 87–152, Nov. 1992.

[12] E.M. Clarke and E.A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," *Proc. Logics of Programs Workshop,* vol. 131, pp. 52–71, 1982.

[13] K.L. McMillan, *Symbolic Model Checking.* Kluwer Academic, 1993.

[14] J. Crow and B. Di Vito, "Formalizing Space Shuttle Software Requirements: Four Case Studies," *ACM Trans. Software Eng. and Methodology,* vol. 7, pp. 296–332, July 1998.

[15] M.P.E. Heimdahl and N.G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Trans. Software Eng.,* vol. 22, no. 6, pp. 363–377, June 1996.

[16] T. Sreemani and J.M. Atlee, "Feasibility of Model Checking Software Requirements: A Case Study," *Proc. 11th Ann. Conf. Computer Assurance (COMPASS '96),* pp. 77–88, June 1996.

[17] J.J. Britt, "Case Study: Applying Formal Methods to the Traffic Alert and Collision Avoidance System (TCAS II)," *Proc. Ninth Ann. Conf. Computer Assurance (COMPASS '94),* pp. 39–51, June/July 1994.

[18] N.G. Leveson, M.P.E. Heimdahl, and J.D. Reese, "Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future," *Proc. Software Eng.—ESEC/FSE '99, Seventh European Software Eng. Conf. Held Jointly with the Seventh ACM SIGSOFT Symp. Foundations of Software Eng.,* O. Nierstrasz and M. Lemoine, eds., pp. 127–145, Sept. 1999.

[19] L. Lamport, "What Good Is Temporal Logic?" *Information Processing 83: Proc. IFIP Ninth World Computer Congress,* R.E.A. Mason, ed., pp. 657–668, Sept. 1983.

[20] M.C. Browne, E.M. Clarke, and O. Grümberg, "Characterizing Finite Kripke Structures in Propositional Temporal Logic," *Theoretical Computer Science,* vol. 59, pp. 115–131, July 1988.

[21] I. Beer, S. Ben-David, and A. Landver, "On-the-Fly Model Checking of RCTL Formulas," *Proc. Computer Aided Verification, 10th Int'l Conf. (CAV '98),* A.J. Hu and M.Y. Vardi, eds., pp. 184–194, June/July 1998.

[22] H. Iwashita, T. Nakata, and F. Hirose, "CTL Model Checking Based on Forward State Traversal," *Proc. Int'l Conf. Computer-Aided Design (ICCAD '96),* pp. 82–87, 1996.

[23] A.J. Hu and D.L. Dill, "Efficient Verification with BDDs Using Implicitly Conjoined Invariants," *Proc. Fifth Int'l Conf. Computer Aided Verification (CAV '93),* C. Courcoubetis, ed., pp. 3–14, June/July 1993

[24] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. Computer-Aided Design,* vol. 13, pp. 401–424, Apr. 1994.

[25] D. Geist and I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions," *Proc. Sixth Int'l Conf. Computer Aided Verification (CAV '94),* D.L. Dill, ed., pp. 299–310, June 1994.

[26] R.K. Ranjan, A. Aziz, R.K. Brayton, B. Plessier, and C. Pixley, "Efficient BDD Algorithms for FSM Synthesis and Verification," *Proc. IEEE/ACM Int'l Workshop Logic Synthesis,* May 1995.

[27] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting Symmetry in Temporal Logic Model Checking," *Formal Methods in System Design,* vol. 9, pp. 77–104, Aug. 1996.

[28] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Partial-Order Reduction in Symbolic State Space Exploration," *Proc. Ninth Int'l Conf. Computer Aided Verification (CAV '97),* O. Grumberg, ed., pp. 340–351, June 1997.

[29] G. Cabodi, P. Camurati, and S. Quer, "Improved Reachability Analysis of Large Finite State Machines," *Proc. Int'l Conf. Computer-Aided Design (ICCAD '96),* pp. 10–14, 1996.
[30] K. Ravi and F. Somenzi, "High-Density Reachability Analysis," *IEEE/ACM Int'l Conf. Computer-Aided Design, Digest of Technical Papers (ICCAD '95)*, pp. 154–158, Nov. 1995.
[31] G. Cabodi, P. Camurati, and S. Quer, "Efficient State Space Pruning in Symbolic Backward Traversal," *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors (ICCD '94),* pp. 230–235, Oct. 1994.
[32] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer, "Disjunctive Partitioning and Partial Iterative Squaring: An Effective Approach for Symbolic Traversal of Large Circuits," *Proc. 34th Design Automation Conf.*, pp. 728–733 June 1997.
[33] B. Yang, R. Simmons, R.E. Bryant, and D.R. O'Hallaron, "Optimizing Symbolic Model Checking for Invariant-Rich Models," *Proc. Computer Aided Verification, 11th Int'l Conf. (CAV '99),* N. Halbwachs and D. Peled, eds., July 1999.
[34] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Conf. Record Fourth ACM Symp. Principles of Programming Languages,* pp. 238–252, Jan. 1977.
[35] E.M. Clarke, O. Grumberg, and D.E. Long, "Model Checking and Abstraction," *ACM Trans. Programming Languages and Systems,* vol. 16, pp. 1512–1542, Sept. 1994.
[36] M.P.E. Heimdahl and M.W. Whalen, "Reduction and Slicing of Hierarchical State Machines," *Proc. Software Eng.—ESEC/FSE '97: Sixth European Software Eng. Conf. Held Jointly with the Fifth ACM SIGSOFT Symp. Foundations of Software Eng.,* M. Jazayeri and H. Schauer, eds., pp. 450–467, Sept. 1997.
[37] R. Bharadwaj and C. Heitmeyer, "Model Checking Complete Requirements Specifications using Abstraction," *J. Automated Software Eng.,* vol. 6, Jan. 1999.
[38] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.,* vol. 10, pp. 352–357, July 1984.
[39] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Trans. Programming Languages and Systems,* vol. 12, pp. 26–60, Jan. 1990.
[40] J. Chang and D. Richardson, "Static and Dynamic Specification Slicing," *Proc. Fourth Irvine Software Symp.,* Apr. 1994.
[41] *IEEE/ACM Int'l Conf. Computer-Aided Design, Digest of Technical Papers,* Nov. 1996.

William Chan completed the PhD degree in the Department of Computer Science and Engineering at the University of Washington in 2000. Growing up in Hong Kong, he went to the US to study in 1991, and received the BS degree with distinction from the Department of Computer Science at Cornell University in 1994. His research interests were in formal methods, particularly, formal verification and symbolic model checking for software systems. He was a student member of the IEEE and a member of the IEEE Computer Society. One week after defending his dissertation and one month before starting as an assistant professor at Brown University, Dr. Chan was killed in a tragic automobile accident.

Richard J. Anderson received the BA degree in mathematics from Reed College in 1981 and the PhD degree in computer science from Stanford University in 1986. Prior to joining the University of Washington in 1986, he was a postdoctoral research fellow at the Mathematical Sciences Research Institute in Berkeley. He is a professor and associate chair in the Department of Computer Science and Engineering at the University of Washington. Dr. Anderson received the US National Science Foundation Presidential Young Investigator Award in 1987 and an Indo-American Fellowship Award in 1993 to support a year-long visit to the Indian Institute of Science in Bangalore, India. His research interests span the field of applied algorithms, including collaborative ventures in astrophysical simulation, symbolic model checking, and web-based typography.

Paul Beame received the BSc degree in 1981, the MSc degree in 1982, and the PhD degree in 1987, all from the University of Toronto. Prior to joining the faculty of the University of Washington, he spent a year as a postdoctoral research associate at the Massachusetts Institute of Technology. Dr. Beame is a professor in the Department of Computer Science and Engineering at the University of Washington, joining the faculty in 1987. He received the US National Science Foundation Presidential Young Investigator Award in 1988, is an associate editor of *Computational Complexity*, and has served on the Steering Committee of the DIMACS Special Year on Logic and Algorithms and the Advisory Committee for the Fields Institute Special Half Year on Complexity. His research interests are in computational complexity, particularly the complexity of propositional proofs, and in the application of computational complexity to problems of formal verification.

David H. Jones graduated from Stanford University (BA, 1969) and Southern Polytechnic State University (AD, 1975). He then proceeded to designed industrial control systems in Atlanta, Georgia. He then moved to Paris, where he led efforts in several companies to introduce new software engineering methods and tools for real-time systems development. He was a consultant in software engineering, company lead on several European research projects, and an instructor at the American College of Paris. He currently does applied research at Boeing Phantom Works, Mathematics and Computing Technology in Bellevue, Washington. Since coming to Boeing in 1988 he has been a technical lead on the DARPA STARS program, applied model-based system development tools on Boeing commercial airplane programs, and investigated how automatic verification technologies can be applied in different areas of Boeing. His other research interests include metadata/model management and knowledge discovery.

David Notkin the ScB degree at Brown University in 1977 and the PhD degree at Carnegie Mellon University in 1984. He is the Boeing Professor of Computer Science and Engineering at the University of Washington. Dr. Notkin received the US National Science Foundation Presidential Young Investigator Award in 1988; served as the program chair of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering; served as program cochair of the 17th International Conference on Software Engineering; chaired the steering committee of the International Conference on Software Engineering (1994-1996); served as a charter associate editor of both *ACM Transactions on Software Engineering and Methodology* and the *Journal of Programming Languages*; serves as an associate editor of the *IEEE Transactions on Software Engineering*; was named as an ACM Fellow in 1998; serves as the chair of ACM SIGSOFT; and received the 2000 University of Washington Distinguished Graduate Mentor Award. His research interests are in software engineering in general and software evolution in particular. Dr. Notkin is a senior member of the IEEE.

William E. Warner received the BS degree in mechanical engineering from the University of Washington in 1983, after which he worked for four years as a nuclear engineer before joining Boeing in 1987. He is currently a systems engineer in the Airplane Systems Laboratory at Boeing Commercial Airplanes. At Boeing, he worked on the 747-400 airplane as an environmental control systems engineer and later as an avionics engineer during development of the 777 airplane. His professional expertise includes requirements analysis and validation, systems modeling and simulation, and software quality assurance.