Improving Efficiency of Symbolic Model Checking for State-Based System Requirements

William Chan Richard J. Anderson Paul Beame David Notkin

{wchan,anderson,beame,notkin}@cs.washington.edu

Department of Computer Science and Engineering

University of Washington, Box 352350

Seattle, WA 98195-2350, USA

Abstract

We present various techniques for improving the time and space efficiency of symbolic model checking for system requirements specified as synchronous finite state machines. We used these techniques in our analysis of the system requirements specification of TCAS II, a complex aircraft collision avoidance system. They together reduce the time and space complexities by orders of magnitude, making feasible some analysis that was previously intractable. The TCAS II requirements were written in RSML, a dialect of statecharts.

Keywords Formal verification, symbolic model checking, reachability analysis, binary decision diagrams, partitioned transition relation, statecharts, RSML, TCAS II, system requirements specification, abstraction.

1 Introduction

Formal verification based on state exploration can be considered an extreme form of simulation: *every* possible behavior of the system is checked for correctness. Symbolic model checking [**?**] using binary decision diagrams (BDDs) [**?**] is an efficient state-exploration technique for finite state systems; it has been successful on verifying (and falsifying) many industry-scale hardware systems. Its application to non-trivial 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 software avionics system for collision avoidance [?]. The full requirements, comprising about four hundred pages, were written in the Requirements State Machine Language (RSML) [**?**], a hierarchical state-machine language variant of statecharts [**?**].

This work was supported in part by National Science Foundation grant CCR-970670. W. Chan was supported in part by a Microsoft graduate fellowship.

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 depend on the structure of the system analyzed. Considerable effort on hardware formal verification has 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 paper, we present modifications to the algorithms implemented in a symbolic model checker (SMV [**?**]), modifications to the model, as well as a simple abstraction technique, to improve the time and space efficiency of the TCAS II analysis. Experimental results show that the techniques together reduce the time and space complexities by orders of magnitude; these improvements have made feasible some analysis that was previously intractable.

The specific techniques we discuss in the paper are:

- *Short-circuiting* to reduce the *number of BDDs* generated by stopping the iterations before a fixed point is reached.
- *Managing forward and backward traversals*, to reduce the *size of the BDD* generated at each iteration. Notably, we improve backward traversals by making certain invariants (in particular, that some events are mutually exclusive) explicit in the search.
- More sophisticated *conjunctive partitioning* of the transition relation and applying *disjunctive partitioning* in an unusual way, to reduce the size of the *intermediate* BDDs at each iteration. Further improvements were made by combining the two techniques to obtain *DNF partitioning*.
- *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.

Techniques like short-circuiting and abstraction are conceptually straightforward and applicable to many systems. Most other techniques were designed to exploit the simple synchronization patterns of TCAS II (for example, most events are mutually exclusive, and most state machines are not enabled simultaneously), and we believe they can also help analyze other statecharts machines with simple synchronization patterns.

We provide experimental results showing how each of these techniques affected the performance of the TCAS II analysis.

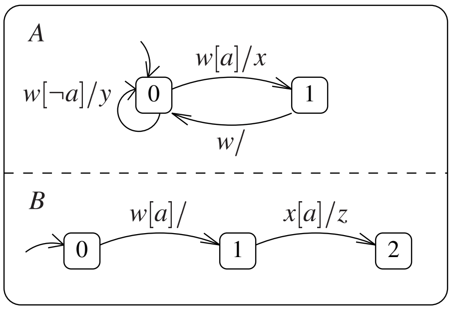


Figure 1: An example of statecharts

The effects of combinations of the improvements are shown in addition to the individual effects. We focus on reachability problems, because most properties of TCAS II we were interested in fall into this class. However, in principle, all of the techniques should benefit general temporal-logic model checking as well. We conclude the paper with discussion on some related techniques.

2 Background

In this section, we give a brief overview of statecharts and RSML. We then turn our attention to symbolic model checking. Finally, we review how we applied symbolic model checking to the TCAS II requirements.

2.1 RSML and Statecharts

The TCAS II requirements were written in RSML, a language based on statecharts. Like other variants of statecharts, RSML extends ordinary state-machine diagrams with state hierarchies; that is, every state can contain orthogonal or mutually exclusive child states. However, this feature does not concern us in this paper (the state hierarchy in the portion of TCAS II that we analyzed is shallow and does not incur special difficulties in model checking). Instead, we can think of the system consisting of a number of parallel state machines, communicating and executing in a *synchronous* way.

Figure **??** above gives a simple example with two parallel state machines A and B. If A is in local state 0, we say that the system is in state $A \triangleright 0$. State machines are synchronized using *events*. Arrows without sources indicate the start local states. Other arrows represent local transitions, which are labeled with the form $u[c]/v$ where u is a *trigger* event, c is the *guarding condition* and v is an *action* event. The guarding condition is simply a predicate on local states of other states machines and/or *inputs* to the system; for example, a guarding condition may say that the system is in $B \triangleright 0$ and an input *altitude* is at least 1 000 meters. (In RSML, the guarding condition is specified separate from the diagram in a tabular form called AND/OR table, but we use the simpler statecharts notation instead.) The guarding condition and the action are optional. The general idea is that, if event u occurs and the guarding condition c either is absent or evaluates to true, then the transition is enabled.

Initially some *external events* along with some (possibly numeric) inputs from the environment arrive, marking the beginning of a *step*. The events may enable some transitions as described above. A *maximal* set of enabled transitions, collectively called a *microstep*, is *taken*—the system leaves the source local states, enters the target local states, and generates the action events (if any). All events are broadcast to the entire system, so these generated events may enable more transitions. The events disappear after one microstep, unless they are regenerated by other transitions. The step is finished when no transitions are enabled. The semantics of RSML assume the *synchrony hypothesis*: During a step, 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.

In Figure ??, assume that w is the only external event, a is a Boolean input, and the system is currently in $A \triangleright 0$ and $B \triangleright 0$. When w arrives, if the input a is false, then the event y is generated. The step is finished since no new transitions are enabled. If instead a is true when w arrives, the transitions from $A \triangleright 0$ to $A \triangleright 1$ and from $B \triangleright 0$ to $B \triangleright 1$ 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 a must be true as before and the external event w cannot occur. So only the transition from $B \triangleright 1$ to $B \triangleright 2$ is enabled and taken, generating event z and finishing the step.

Subtle but important semantic differences exist among variants of statecharts. The semantics of STATEMATE [**?**], another major variant of statecharts, are close to those of RSML. STATEMATE does not enforce the synchrony hypothesis in the semantics, but provides it as an option in the simulator. RSML and STATEMATE also have a richer set of synchronization primitives and provide some sort of variable assignments; however, these features are not important for this paper.

2.2 Symbolic Model Checking

We now switch gears to discuss ordinary finite-state transition systems (without state hierarchies, the synchrony hypothesis, etc.) and model checking. The goal of model checking is to determine whether a given state transition system satisfies a property given as a temporal logic formula, and if not, try to give a counterexample (a sequence of states that falsifies the formula). Example properties include that a (bad) state is never reached, and that a (good) state is always reached infinitely often. In "explicit" model checking, the answer is determined in a graph-theoretic manner by traversing and labeling the vertices in the state graph [**?**]. 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 *sets* of states instead of individual states.

For illustration, we focus on the reachability problem, the simplest and the most common kind of temporal property checked in practice. Let Q be the finite set of system states, $R \subseteq Q \times Q$ the state transition relation, $I \subseteq Q$ the set of initial states, and $E \subseteq Q$ a set of error states. The reachability problem asks whether the system always stays away from the error states E, and if not, demands a counterexample, that is, a sequence of states $q_0, q_1, \ldots, q_n$ with $q_0 \in I$, $q_n \in E$ and $(q_i, q_{i+1}) \in R$ for $0 \leq i < n$.

We define $Pre\colon 2^Q \mapsto 2^Q$ to compute the *pre-image* (or the

Start with $Y_0 = E$ and iteratively compute $Y_{i+1} = Pre(Y_i) \cup Y_i$ until reaching a fixed point.

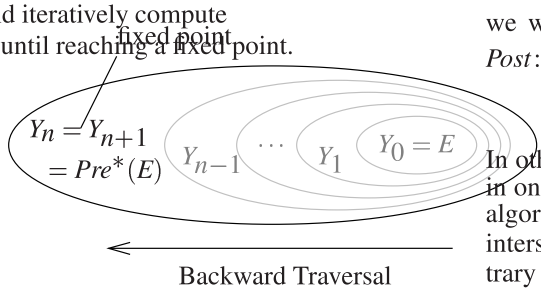
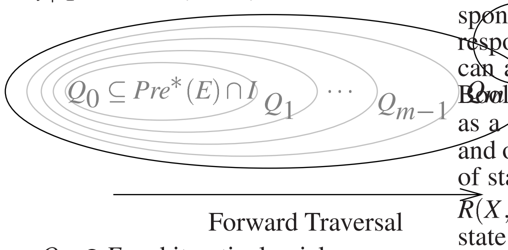


Figure 2: An algorithm for computing $Pre^*(E)$

1. Let Q_0 be any nonempty subset of $Pre^*(E) \cap I$. Iteratively compute $Q_{i+1} = Post(Q_i) \cup Q_i$ until reaching E.



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

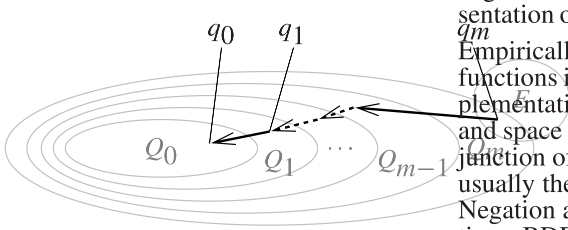


Figure 3: An algorithm for counterexample search

weakest pre-condition) of a set of states under the transition relation R:

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

Intuitively, it is the set of states that may reach some state in S in one transition. Then we can characterize the decision problem of reachability in a set-theoretic manner using *fixed points*: Determine whether $I \cap Pre^*(E)$ is empty, where $Pre^*(E)$ is the set of states that may eventually reach an error state. More specifically, it is the smallest state set Y that satisfies

$$Y = E \cup Pre(Y).$$

Its existence is guaranteed by the finiteness of Q and the monotonicity of *Pre*. Figure **??** shows an iterative algorithm for computing this fixed point. The set Y_i is the states that may reach an error state in at most i transitions. Many other temporal properties can be similarly defined and computed using (possibly multiple or nested) fixed points [?].

If the intersection of $Pre^*(E)$ and the initial states I is empty, then the set E is not reachable and we are done. Otherwise, we would like to find a counterexample. We first define $Post\colon 2^Q \mapsto 2^Q$ to compute *post-images*:

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

In other words, $Post(S)$ is the set of states reachable from S in one transition. Figure **??** shows a counterexample search algorithm. The set Q_0 can be any nonempty subset of the intersection, but it is convenient to choose Q_0 to be an arbitrary singleton set. The set Q_i is the states that are reachable from Q_0 in at most i transitions. We obtain a counterexample by tracing backward from $Q_m \cap E$. (We will improve this algorithm later.)

The crucial factor for efficiency is the representation for state sets. Notice that the state space Q can be represented by a finite set of variables X, such that each state in Q corresponds to a valuation for the variables and no two states correspond to the same valuation. For finite state systems, we can assume without loss of generality that each variable is Boolean. A set of states S is then *symbolically* represented as a Boolean function $S(X)$ such that a state is in the set if and only if it makes the function true. The transition relation of states can be similarly represented as a Boolean function $R(X,X')$ where X' is a copy of X and represents the next state. Intersection, union and complementation on sets or relations respectively becomes conjunction, disjunction and negation on Boolean functions. Now the problem of representation of state sets is reduced to that of Boolean functions. Empirically, the most efficient representation for Boolean functions is BDDs [?]. They are canonical, with efficient implementation for Boolean operations. For example, the time and space complexities of computing the conjunction or disjunction of two BDDs are at most the product of their sizes; usually the complexities observed in practice are still lower. Negation and equivalence checking can be done in constant time. BDDs are often succinct, but this relies critically on a chosen linear *variable order* of the variables in X.

We can now represent a state set S and the transition relation R as BDDs and compute the pre-image and post-image of S as follows:

$$\begin{aligned} Pre(S(X')) &= \exists X'.\, R(X,X') \wedge S(X'), \\ Post(S(X)) &= \exists X.\, R(X,X') \wedge S(X). \end{aligned}$$

The notation $\exists X$ refers to existentially quantifying out all the variables in X. In addition to Boolean operations and equivalence checking, operations like existential quantification and variable substitution can also be performed, so the algorithms in Figures ?? and **??** (and similar algorithms for many temporal logics such as CTL [?]) can be implemented using BDDs. Thanks to the succinctness of BDDs and the efficiency of their algorithms, some systems with over 10^{120} states can be analyzed [?].

2.3 Symbolic Model Checking for TCAS II

We analyzed the TCAS II requirements using a symbolic model checker SMV (Version 2.4.4). SMV uses algorithms similar to those in Figures ?? and **??**. A notable difference is that in Figure ??, instead of computing $Y_{i+1} = Pre(Y_i) \cup Y_i$, it uses the equivalent recurrence $Y_{i+1} = Pre(Y_i - Y_{i-1}) \cup Y_i$, with the advantage that $Y_i - Y_{i-1}$ usually requires a much smaller BDD than Y_i does, resulting in faster pre-image computation. (In fact, it is sufficient to compute the pre-image of any Z with $Y_i - Y_{i-1} \subseteq Z \subseteq Y_i$ [**?**].) Similar comments apply to the computation of each Q_i in Figure **??**.

Because SMV does not support hierarchical states and other RSML features directly, we had to translate the requirements into an ordinary finite-state transition system in the SMV language. The requirements consist of two main parts, Own-Aircraft and Other-Aircraft, which occupy about 30% and 70% of the document respectively. In our initial study, we translated Own-Aircraft quite faithfully to the SMV language, and abstracted Other-Aircraft as a mostly nondeterministic state machine. The details of the translation, including how the transitions, the state hierarchy and the synchrony hypothesis were handled, as well as the properties analyzed, were given in a previous paper [?]. Certain details about the system model are relevant to this paper:

- An RSML microstep corresponds to a transition in the SMV program, and thus a step corresponds to a sequence of transitions.
- We encode each RSML event as a Boolean variable, which is true if and only if the event has just occurred.
- We assume each numeric input to be discrete and bounded, and encode each *bit* as a Boolean variable.
- To maintain the synchrony hypothesis, we prevent the inputs from changing and the external events from arriving when some of the variables that encode events are true.
- We analyze one instance of TCAS II only, so the asynchrony among multiple instances of the system is not an issue.

A major source of complexity of the analysis was the transitions' guarding conditions, some of which occupy many pages of description. They contain predicates of local states and of the input variables, and may involve complex arithmetic. While many other researchers conservatively encode each arithmetic predicate as an independent Boolean variable [**?**, **?**, ?], we encode each input bit as a Boolean variable, resulting in more accurate analysis at the expense of 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 Own-Aircraft is simple, and concurrency among the state machines in Own-Aircraft is minimal. As we will see, some of the techniques presented later attempt to exploit these simple synchronization patterns.

3 Short-Circuiting

It is easy to see that in Figure ??, we do not need to compute a fixed point when the error states are reachable—we can stop once the intersection of some Y_i and I is not empty, because all we need is an element in the intersection. This *short-circuiting* technique may substantially reduce the time and space used when a short counterexample exists.

More generally, short-circuiting can be applied to the outermost temporal operator in temporal-logic model checking (however, the reduction obtained is probably less than

Start with some $q_0 \in Y_n \cap I$ and iteratively pick some $q_i \in Post(q_{i-1}) \cap Y_{n-i}$ to obtain a counterexample q_0, q_1, $\ldots$, q_n.

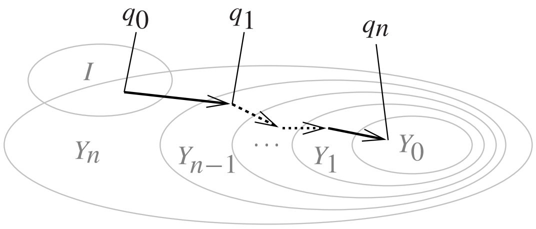


Figure 4: A simplified algorithm for counterexample search

in reachability analysis, because only one of the many fixed points can be stopped prematurely.)

4 Forward vs. Backward Traversals

Fixed-point computation or counterexample search can be done either forward or backward. In this section we elaborate on their performance difference in our analysis. In short, backward traversals generate smaller BDDs and are a big win for our system. They can be further improved by incorporating certain invariants to prune the searches.

4.1 Improved Counterexample Search

During the analysis of TCAS II, we found that when a property was disproved in a few minutes, finding a counterexample might take hours. A coauthor of a previous paper subsequently simplified the counterexample search algorithm, resulting in substantial speedup [?]. This is the only technique described here that was used in that study.

The forward traversal in the first part of Figure **??** is the bottleneck. For our system, the sequence of post-images requires large BDDs. However, we can eliminate this step if we remember every Y_i computed in Figure ?? (our actual implementation stores the difference $Y_i - Y_{i-1}$ instead of Y_i). Our modification, illustrated in Figure **??**, is by no means innovative and should be considered natural.[1] A disadvantage of the algorithm is the use of additional memory to store the state sets, which is wasted in case the error states are not reachable. Nevertheless, the dramatic speedup made possible far outweighs the modest additional memory requirements.

An important question remains: Why is the backward traversal in Figure ?? much more efficient than the forward traversal in Figure **??**? The inefficiency of forward traversals is also witnessed by SMV's inability to compute the set of reachable states of the system. Finding the reachable state set by searching forward from the initial states is a common technique in hardware verification; the set can be used to help analyze other temporal properties and synthesize the

[1] Indeed, if we search forward to find the reachable state set, SMV can optionally use a similar counterexample search algorithm, but it is not used with the default backward traversal.

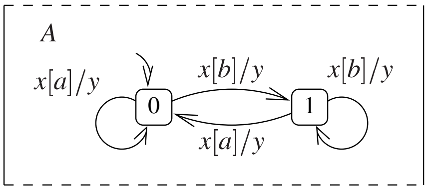


Figure 5: A state machine with local invariants

circuit.

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 general, we observe that in backward traversals, the BDDs 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. Nevertheless, the verification of many hardware systems tends to benefit, rather than suffer, from forward traversals. For example, Iwashita et al. report significant speedup in CTL model checking for their hardware benchmarks when forward instead of backward traversals are used [**?**].

Partly inspired by Hu and Dill [**?**], we believe that the inefficiency is mainly due to the complex invariants of TCAS II, which are maintained by forward but not backward traversals. As an example, consider the state machine in Figure **??**. If event y is only generated in A, then an invariant of the system is that, whenever event y has just occurred, the machine is in $A \triangleright 0$ if and only if condition a is true. If the BDD for a is large, so will the BDD for the invariant. Even if they are small, there are likely to be many such implicit invariants in the system, and their conjunction may have a large BDD representation. In addition, invariants may globally relate different state machines, 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 low-level hardware verification, the BDDs often remain small, because each invariant is usually localized and involves only a small number of state variables. This is however not the case in TCAS II.

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 a does not imply 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 their BDDs tend to be smaller for our system.

4.2 Improved Backward Traversals Using Invariants

Interestingly, the main disadvantage of backward traversals is also that (forward) invariants are not maintained. Some invariants, particularly those with small BDDs, can help simplify the BDDs of state sets, and can speed up backward traversals if they are incorporated into the search. In the context of statecharts, many systems have simple synchronization patterns, which are lost in backward traversals. A particular invariant that we find useful to rectify this problem is the *mutual exclusion of certain events*. We illustrate this idea with an example.

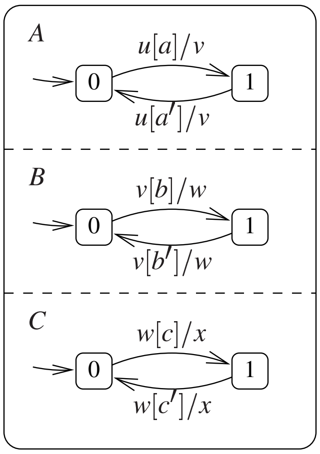


Figure 6: A system with a linear structure

Consider the system in Figure **??**. Assuming u is the only external event, there is no concurrency in the system—at most one local transition can be enabled at any time. Forward traversals do not explore concurrent executions of the state machines.

However, in backward traversals, the analysis may be fooled to consider many concurrent executions, which are not reachable. Suppose we want to check whether the system can be in $B \triangleright 1$ and $C \triangleright 1$ simultaneously. Traversing backward, we find that in the previous microstep, the system may be in $(B \triangleright 0, C \triangleright 1)$, $(B \triangleright 1, C \triangleright 0)$, or $(B \triangleright 0, C \triangleright 0)$. The last case, however, is not possible, because events v and w cannot occur at the same time. (Notice that this is true only if we assume the synchrony hypothesis.) Tracing more iterations, we can see that the search considers not only concurrent executions but also many unreachable interleaving ones. The BDDs thus may blow up if the guarding conditions are complex.

Fortunately, we can greatly simplify the search by observing that all the events are mutually exclusive. This invariant can be incorporated into the traversals by either intersecting it with the state sets or using it as a care-set to simplify the BDDs [?].

To find out such a set of mutually exclusive events, we may perform a conservative static analysis on the causality of the events. Alternatively, the designer may know which events are mutually exclusive, because the synchronization patterns should have been designed under careful consideration. To confirm the mutual exclusion, we may verify, using model checking or other static analysis techniques, that the states with

$$\bigvee_{\substack{u,v \in \Sigma \\ u \neq v}} (u \wedge v)$$

are not reachable, where Σ is the set of state variables encoding the events under consideration. In the case of TCAS II, a large part of our model behaves similarly to the machine in Figure **??**, and the set of mutually exclusive events was evident.

5 Partitioned Transition Relation

Apart from the BDD size for state sets, another bottleneck of model checking is the BDD size for the transition relation, which can be reduced by *conjunctive* or *disjunctive partitioning* [?]. The former can be used naturally for TCAS II, 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 image computation.

5.1 Background

In this subsection, we review the idea of conjunctive and disjunctive partitioning, described in Burch et al. [?]. 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 image computations can be easily modified by distributing the existential quantification over the disjunction. For pre-image computation, we thus have

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

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

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

So we can compute the pre-image without ever building the BDD for R. Post-image computation is symmetric.

If, however, 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 image 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 the 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 $p < i$. Consider again pre-image computation. We compute

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

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 $C_1, C_2, \ldots, C_k$.

5.2 Determining Conjunctive Partition

We could not construct the monolithic BDD for the transition relation R for our model of TCAS II in hours of CPU time, but R 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. SMV 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 SMV to allow the user to specify the partition manually. We also implemented in SMV a variant of the heuristics by Geist and Beer [**?**] and by Ranjan et al. [**?**] to automatically determine the partition. The central idea behind the heuristics is to 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.

5.3 Disjunctive Partitioning for Statecharts

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. (Another advantage that we have not exploited is the possibility of parallelizing the image computation by constructing the intermediate BDDs concurrently.)

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* $\alpha_1(X,X'), \alpha_2(X,X'), \ldots, \alpha_j(X,X')$ such that their disjunction is the tautology, then we can indeed disjunctively partition R by distributing R over the cover:

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

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

$$\begin{aligned} D_i(X,X') &= \alpha_i(X,X') \wedge R(X,X') \\ &= \alpha_i(X,X') \wedge C_1(X,X') \wedge C_2(X,X') \wedge \cdots \wedge C_k(X,X'). \end{aligned}$$

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

For TCAS II and many other statecharts, however, we can again exploit the mutual exclusion of certain events, say u_1, $u_2, \ldots, u_{j-1}$. Define

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

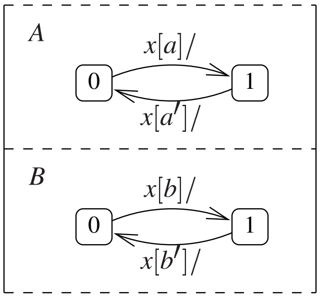


Figure 7: Event x triggers two state machines.

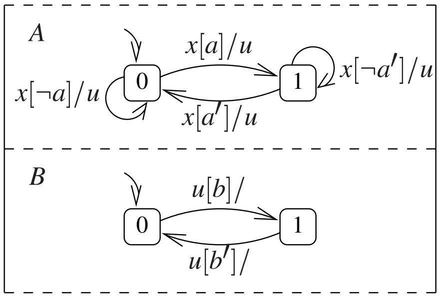


Figure 8: The serialized machine

for $1 \leq i < j$, and

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

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, most of the parallel assignments in our SMV program are guarded by conditions on the events; for example, an assignment that models a state transition requires the occurrence of the trigger event. If the event is, say u_i for some $1 \leq i < j$, then the BDD for the assignment is applicable only to the disjunct D_i, and all the other disjuncts of the transition relation are unaffected. 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 **??**.

5.4 DNF Partitioning and Serialization

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

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

Alternatively, we may serialize the complicated microstep into cascading microsteps to reduce the BDD size. Figure **??** 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, 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 the machine B do not refer to machine A's local states, x is mutually exclusive with all other events, and we are checking a reachability property that does not explicitly mention any of the state machines, transitions or events involved in the transformation.[2]

6 Abstraction

In this section, we give a simple algorithm to remove part of the system from the model that is guaranteed not to interfere with the property being checked. For example, a state machine may have a number of outputs (which may be local states or events). If we are verifying only one of them, the logic that produces other outputs may be abstracted away, provided these outputs are not fed back to the system. The abstraction obtained is *exact* with respect to the property, in the sense that the particular property holds in the abstracted model if and only if it holds in the original model.

6.1 Dependency Analysis

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

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

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. It should be evident that everything not determined relevant by these rules can be removed without affecting the analysis result.

[2] The same criterion can be applied to arbitrary CTL formulas, provided we do not use the the next-time operator X, which can count the number of microsteps. In other words, under the assumptions, the transformation preserves equivalence under stuttering bisimulation [**?**].

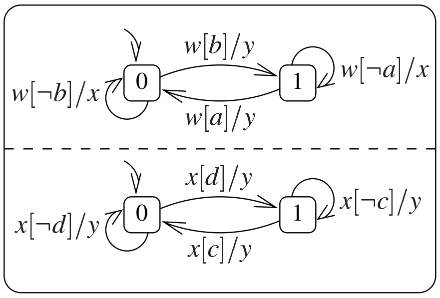


Figure 9: False dependency: Event *y* does not depends on any guarding condition.

6.2 False Dependency

Similar dependency analyses could also be performed by model checkers (such as VIS [**?**]) on the Boolean model of the statecharts machine. However, a straightforward implementation would not be effective. The reason is that in the model, an input would appear to depend on every event because of the way we encoded the synchrony hypothesis (Section **??**). On the other hand, carrying out dependency analysis on the high-level statecharts description does not fall prey to such false dependencies.

Other forms of false dependencies are possible, however. Suppose we are given the system in Figure **??** 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 [?]. However, the syntax of RSML and STATEMATE allows easy detection of most false dependencies of this kind. Notice that the self-loops in Figure **??** are solely for synchronization—they make sure that u is generated regardless whether there has been a local state change. To improve the visual presentation, RSML and STATEMATE allow specifying the generation of such events separate from the state diagram using *identity transitions* and *static reactions* respectively. (Actually, their semantics are slightly different from self-loops, but the distinctions are not important here.)

Some false dependencies are harder to detect automatically. For example, maybe the guarding conditions involved do not form a tautology, but in all *reachable* states, one of the guarding conditions holds whenever the trigger event occurs. As another example, in Figure **??**, the event y does not depends on any of the guarding conditions, because it is always generated one or two microsteps after w.[3] 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 abstracted model of the system. In our TCAS II analysis, the synchronization of Own-Aircraft is simple enough that false dependencies can be easily detected. However, this method may be used for analyzing the rest of TCAS II or other systems.

[3] However, if the next-time operator X is used, then y may be considered conservatively to be dependent on a and b.

7 Experimental Results

The table above summarizes the results of applying the techniques mentioned to our model 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 six properties. Note that the latter excludes the time spent on building the transition relation or the resources for finding the counterexamples. The counterexample search took about one to two seconds per state in the counterexample and was never a bottleneck thanks to the algorithm in Figure **??**. That algorithm was used in all the checks, because without it, none of the counterexamples could be found in less than one hour. The table also shows the number of iterations needed to reach fixed points and the length of the (shortest) counterexamples. We performed the experiments on a Sun SPARCstation 10 with 128MB of main memory. Most successful checks used less than 30MB of main memory.

Properties P1 through P4 refers to the properties Increase-Descent Inhibition, Function Consistency, Transition Consistency, and Output Agreement explained in a previous paper [?]. Property P5 refers to an assertion in Britt [**?**, p. 49] that Own-Aircraft should never be in two local states Corrective-Climb ▷ Yes and Corrective-Descend ▷ Yes simultaneously (comments in our version of the TCAS II requirements, however, explicitly say that the two local states are not mutually exclusive). Property P6 is somewhat contrived: It is simply the conjunction of P3 and P4. Since searching simultaneously from two unrelated sets of states tends to blow up the BDDs, checking this property provides an easy way of scaling up the BDD size. It also mimics checking properties involving a large part of the system. All six properties are reachability, and are violated by the model. An entry with ∞ in the table indicates timeout after one hour.

We emphasize that the purpose of the data is to investigate the general effects of the techniques on our model of TCAS II. They are not for picking a clear winner among the techniques, since the BDD algorithms are very sensitive to the various parameters chosen and to the model analyzed. Note also that the results shown here should not be compared directly with out earlier results [?], because the models, the parameters, and the model checking algorithms used were different.

Full Model The first part of the table shows the results for the full model with 227 Boolean state variables. Row 1 gives the results for the base analysis: Two properties could not be completed using the conjunctive partitioning as implemented in SMV. (Actually, we implemented a small improvement that was used in all results including the base analysis. As explained in Section **??**, an image computation step involves a conjunction and an existential quantification. The two operations can be carried out simultaneously to avoid building the usually large conjunction explicitly [?]. SMV performs this optimization except when conjunctive partitioning is used.

					Building BDDs for R		P1		P2		P3		P4		P5		P6		
Full Model (227 variables)																			
No. of fixpoint iterations							24		29		29		38		26		26		
Counterexample length							15		15		11		24		17		11		
	Optimizations SC	MX	CP	DP	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	
1	—	—	—	—	20	93	79	400	182	713	257	1060	342	1090	∞		∞		1
2	√	—	—	—	20	93	62	400	143	713	61	669	136	751	∞		∞		2
3	—	√	—	—	20	94	11	110	20	123	76	369	38	152	47	249	490	1903	3
4	—	—	√	—	33	176	40	273	97	345	147	488	193	412	∞		∞		4
5	—	√	—	√	34	464	18	464	33	464	798	968	34	463	74	480	5871	1603	5
6	—	√	√	—	25	166	9	170	18	190	51	267	31	215	39	245	316	1139	6
7	—	√	√	√	40	128	7	128	14	139	57	217	24	150	29	160	320	1022	7
8	√	√	√	√	41	128	6	128	8	128	13	153	12	143	18	141	23	243	8
Mistranslated Model† (227 variables)																			
	Optimizations SC	MX	CP	DP	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	
9	—	√	—	—	20	93	285	697	317	1016	95	314	518	1129	615	2245	442	1591	9
10	—	√	√	—	26	174	323	1043	791	1546	91	424	497	1471	∞		871	2186	10
11	—	√	√	√	42	126	126	327	154	515	49	185	215	398	213	678	198	547	11
Serialized Model (231 variables)																			
No. of fixpoint iterations							36		41		45		54		38		38		
Counterexample length							23		23		19		36		25		19		
	Optimizations SC	MX	CP	DP	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	
12	—	√	—	—	27	103	12	111	39	190	127	311	46	144	89	325	867	2307	12
13	—	√	√	—	31	167	12	167	38	234	127	323	44	199	94	363	959	1932	13
14	—	√	—	√	27	139	12	139	40	161	136	251	32	160	76	177	897	1040	14
15	—	√	√	√	48	136	11	136	34	162	129	221	39	156	74	196	762	982	15
Abstracted Models†																			
No. of variables							142		142		150		142		150		171		
	Optimizations SC	MX	CP	DP	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	time (s)	nodes (K)	
16	—	—	—	—	vary		5	65	17	93	72	362	26	115	∞		∞		16
17	√	√	√	√	vary		2	33	4	39	6	73	6	40	13	95	18	158	17

SC: short-circuiting MX: mutual exclusion of events CP: improved conjunctive partitioning DP: disjunctive partitioning

†No. of fixpoint iterations and counterexample lengths are identical to those of the full model.

Table 1: Resources used in the analysis

We simply changed SMV to do this optimization with conjunctive partitioning.)

As expected, short-circuiting (SC) gave savings, because the number of iterations needed became the length of the shortest counterexample. Incorporating the mutual exclusion of certain events into backward traversals (MX) generally gave an order of magnitude time and space reduction. In addition, we could now easily disprove P5 and P6 (in particular, Britt's claim mentioned above is provably not true in our version of the requirements). For improved conjunctive partitioning (CP), as mentioned in Section **??**, we used a heuristic to produce a partition, which was effective in reducing time and space used.

Disjunctive partitioning (DP), which must be combined with mutual exclusion of events, appeared to be inefficient (Row 5). 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 large disjuncts, leading to DNF partitioning (indicated on Row 7 by marking both CP and DP). It performs marginally better than pure conjunctive partitioning in terms of time, but the space requirements were consistently lower. Combining it with the other two optimizations, we observed orders of magnitude improvements in time and space (Row 8).

Mistranslated Model To further illustrate the differences between conjunctive and DNF partitioning, we looked at a version of the model that contains a translation error from the RSML machine to the SMV program. It was a real bug we made early in the study, although we soon discovered it by inspection. The mistake was omitting some self-loops similar to those in Figure **??**. BDDs for faulty systems are often larger than those for the corrected versions, because bugs tend to make the system behavior less "regular". Therefore, investigating the performance of BDD algorithms on faulty designs is meaningful.

Interestingly, the particular partition generated by the the heuristic performed poorly for this model (Row 10). DNF partitioning continued to give significant time and space savings (Row 11).

Serialized Model As mentioned above, the disjunctive partition contains a disproportionally large BDD. We serialized a microstep in the full model to break the large disjunct into four BDDs of sizes about a hundred times smaller. As expected, disjunctive partitioning now performed better (Rows 5 vs. 14). However, since the number of microsteps in a step increased, all partitioning techniques 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.

The data suggest that serializing the microstep in order to use disjunctive partitioning is not advantageous for this model. In general, we find the effects of serializing and its dual, collapsing microsteps, difficult to predict. It represents a tradeoff between the complexity of image computations and the number of search iterations.

Abstracted Models The last part of the table shows the performance of analyzing the abstracted models obtained by the dependency analysis in Section **??**. The number of variables abstracted away is quite large. Recall that in our full model, we omitted most of the details in Other-Aircraft. Many of the outputs of Own-Aircraft that are inputs to Other-Aircraft thus become irrelevant, unless we explicitly mention them in the property. This explains the relatively large reduction obtained.

8 Discussion and Related Work

We first summarize some differences between symbolic model checking for hardware circuits and for TCAS II. A major focus of hardware verification is on concurrent systems with complex control paths and often subtle concurrency bugs, but their data paths are relatively simple. Forward traversals usually perform much better, because the BDDs tend to be small in their reachable state spaces. In contrast, the major complexity of the TCAS II requirements lies not in the concurrency among components, but in the intricate influence of data values on the control paths. The BDD for the transition relation tends to be huge and forward traversals inefficient. Backward traversals usually perform better by focusing on the property analyzed, and can be further improved by exploiting the simple synchronization patterns.

Our method of pruning backward traversals using invariants is similar in spirit to the work on hardware verification by Cabodi et al., who propose doing an *approximate* forward traversal to compute a superset of the reachable states, which is then used to prune backward traversals [**?**]. Their method is more automatic, while the invariants we suggest take advantage of the simple synchronization of the system. They also independently propose disjunctive partitioning for synchronous circuits [**?**]. Their method requires the designer to come up with a partition manually, and we again exploit mutually exclusive events.

In work also independent of ours, Heimdahl and Whalen [**?**] use a dependency analysis technique similar to the one described Section **??**, but their motivation is to facilitate manual review of the TCAS II requirements, rather than automatic verification. As noted before, we gained relatively large reduction because Other-Aircraft was not fully modeled, and we suspect that in a complete system, the reduction obtained by this exact analysis could be limited. However, more reduction can be obtained if we forsake exactness. For example, *localization reduction* [**?**] is one such technique, which aggressively generates an abstracted model that may not satisfy the property while the full model does. If the model checker finds in the abstracted model a counterexample that does not exist in the full model, it will automatically refine the abstraction and iterate the process until either a correct counterexample is found or the property is verified.

It would be interesting to see how well the techniques in this paper scale with the system complexity. The natural way is to try applying them to the rest of TCAS II. Unfortunately, that part contains arithmetic operations, such as multiplication, that provably cannot be represented by small BDDs [**?**]. In a recent paper, we suggest coupling a decision procedure for nonlinear arithmetic constraints with BDD-based model checking to attack the problem [**?**]. More research is needed to see whether this technique scales to large systems.

Acknowledgments

We thank Steve Burns, who observed the inefficiency of the algorithm in Figure **??** and implemented the one in Figure **??** in SMV.

References

[1] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. In D. Garlan, editor, *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 156–166, San Francisco, CA, USA, October 1996.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference, Proceedings 1990*, pages 40–45, Orlando, FL, USA, June 1990.

[3] J. J. Britt. Case study: Applying formal methods to the Traffic Alert and Collision Avoidance System (TCAS) II. In *COMPASS'94, Proceedings of the 9th Annual Conference on Computer Assurance*, pages 39–51, Gaithersburg, MD, USA, June/July 1994. IEEE.

[4] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, August 1986.

[6] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[7] G. Cabodi, P. Camurati, and S. Quer. Efficient state space pruning in symbolic backward traversal. In *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–235, Cambridge, MA, USA, October 1994.

[8] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV'97 Proceedings*, pages 316–327, Haifa, Israel, June 1997. Springer-Verlag.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *11th International Conference on Computer Hardware Description Languages and Their Applications, CHDL'93*, Ottawa, Ontario, Canada, April 1993. IEEE.

[11] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems, International Workshop Proceedings*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[12] M. B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *Proceedings of the Joint 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[14] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[15] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency analysis of state-based requirements. In R. Jeffrey and D. Notkin, editors, *17th International Conference on Software Engineering*, pages 3–14, Seattle, WA, USA, April 1995.

[16] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th ACM/IEEE Design Automation Conference, Proceedings 1993*, pages 266–271, Dallas, TX, USA, June 1993. ACM/IEEE.

[17] A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st ACM/IEEE Design Automation Conference, Proceedings 1994*, pages 276–282, Dallas, TX, USA, June 1994. ACM/IEEE.

[18] H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *1996 IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, pages 82–87, San Jose, CA, USA, November 1996.

[19] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[20] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, SE-20(9), September 1994.

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

[22] K. L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–251, Tokyo, Japan, April 1991.

[23] R. Pugliese and E. Tronci. Automatic verification of a hydroelectric power plant. In M. C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods, 3rd International Symposium of Formal Methods Europe, Proceedings*, pages 425–444, Oxford, UK, March 1996. Springer-Verlag.

[24] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *1993 IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, pages 42–47, November 1993.

[25] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements: A case study. In *COMPASS'96, Proceedings of the 11th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 1996. IEEE.