

Decoupling Synchronization from Local Control for Efficient Symbolic Model Checking of Statecharts

William Chan[†]

Richard J. Anderson[†]

Paul Beame[†]

David H. Jones^{*}

David Notkin[†]

William E. Warner^{*}

[†]Department of Computer Science and Engineering, University of Washington

Box 352350, Seattle, Washington 98195-2350, USA +1-206-543-1695

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

^{*}The Boeing Company, Seattle, Washington, USA

{david.h.jones, william.e.warner}@boeing.com

ABSTRACT

Symbolic model checking is a powerful formal-verification technique for reactive systems. In this paper we address the problem of symbolic model checking for software specifications written as statecharts. We concentrate on how the synchronization of statecharts relates to the efficiency of model checking. We show that statecharts synchronized in an oblivious manner, such that the synchronization and the local control are decoupled, tend to be easier for symbolic analysis. Based on this insight, the verification of some non-oblivious systems can be optimized by a simple, transparent modification to the model to separate the synchronization from the local control. The technique enabled the analysis of the statecharts model of a fault-tolerant electrical power distribution system developed by the Boeing Commercial Airplane Group. The results disclosed subtle modeling and logical flaws not found by simulation.

Keywords

Formal methods, formal verification, symbolic model checking, binary decision diagrams, software specification, statecharts, fault tolerance.

1 INTRODUCTION

Symbolic model checking [4] shows promise as an aid to producing industrial-strength software specifications in which developers have increased confidence [6, 20]. The formal languages for writing such specifications allow developers to produce specifications in a number of different styles. Just as the way that a program is written affects how efficiently one

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

Appeared in the *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, Los Angeles, USA, May 1999.

© Copyright 1999 by ACM, Inc.

can analyze it, the style used to describe a specification affects how efficiently one can analyze it using symbolic model checking.

The statecharts language is one of the most broadly used languages for specifying reactive systems [10]. In this paper, we address how the synchronization in a statecharts specification influences the efficiency of symbolic model checking. We identify certain styles for synchronization that are more efficient for symbolic analysis. 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 work started as a case study of applying symbolic model checking based on binary decision diagrams (BDDs) [3] to a statecharts specification developed by the Boeing Commercial Airplane Group. Previously, the same technique was applied to the requirements specification of the airborne collision avoidance system TCAS II [6, 7] written in the Requirements State Machine Language (RSML) [14], a language also based on statecharts. The observations and the optimization technique described in this paper result from the combined experience of these two case studies.

In symbolic model checking, the state space of a formal model is exhaustively explored. Sets of states are represented implicitly, so the method is not restricted by the state-space size and is able to analyze many systems much larger than conventional techniques can handle. The efficiency relies mainly on the succinctness of the symbolic representations such as BDDs, but their size is usually hard to predict. For software specifications, it depends not only on the functionality of the system but also on the particular way in which the specification is written.

Our model of statecharts responds to environment inputs by performing a macrostep, divided into a number of microsteps synchronized by events. We found that statecharts with events synchronized in an *oblivious* manner, such as the TCAS II requirements, tend to be more amenable to our analysis—the state machines' synchronization is decou-

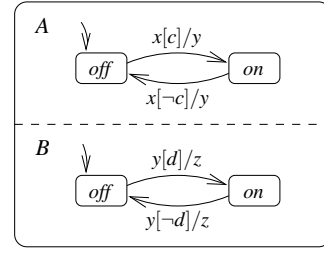
pled from their local control, resulting in fewer dependencies among the state variables and smaller BDDs. However, we observe that the length of a macrostep can often be bounded statically. By artificially incorporating a *microstep counter* into the statecharts specification, we can decouple synchronization and local control in non-oblivious systems as well. The microstep counter also helps prune certain unreachable states from backward searches. The incorporation of the microstep counter is transparent to the specifier, and preserves the model-checking results for most interesting properties (formally, all stutter-invariant properties, including all temporal-logic formulas without the next-time operator, are preserved [13]). The technique is interesting, particularly because it achieved substantial time and space improvements in our case study 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.

Another contribution of this work is the case study itself. Formal models have been used increasingly in Boeing to specify and validate functional requirements of airborne computing systems [16]. One of the modeling languages used is statecharts, thanks to their intuitive notations, ability to scale, and the availability of supported tools [17]. Developed for research purposes, the statecharts studied in this work model a fault-tolerant electrical power distribution (EPD) system designed for use on aircraft. Its purpose is to distribute electrical power from power sources to power busses 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 to 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.

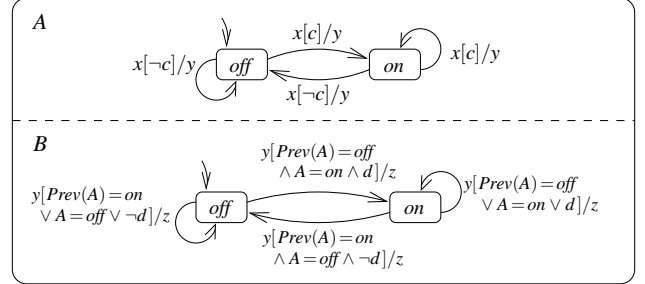
The rest of the paper is organized as follows. We first review statecharts and symbolic model checking in the next two sections. In Section 4 we explain the differences that oblivious and non-oblivious systems make to the efficiency of symbolic model checking. Our optimization technique is presented in Section 5. We describe the model of the EPD system and the results of the analysis in Section 6. Finally, Section 7 concludes the paper with some lessons learned.

2 STATECHARTS

The statecharts language is a popular visual language for specifying complex reactive systems [10]. It extends state-machine diagrams with parallelism, superstates, and broadcast communications. For simplicity, we 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 set of parallel local state ma-



(a) Non-oblivious synchronization



(b) Oblivious synchronization

Figure 1: Two ways to specify machines A and B in statecharts, with equivalent stable-state behaviors

chines, with a finite set of events and inputs, all embedded in a nondeterministic environment.

2.1 Syntax and Semantics

Figure 1(a) gives an example with two parallel state machines A and B and events x , y , and z . Arrows without sources indicate the initial local states. Other arrows represent local transitions, which have labels of the form $trig[cond]/acts$, where $trig$ is a *trigger* event, $cond$ a *guarding condition*, and $acts$ a (possibly empty) list of *action* events. The guarding condition is a predicate on local states of other state machines and/or inputs to the system. A label of the form $trig[true]/acts$ is abbreviated as $trig/acts$. The transition is enabled whenever the machine is in the source local state, the event $trig$ occurs, and the guarding condition $cond$ is true.

An *external* event is one that can be generated by the environment. For simplicity, we assume that an external event cannot also be an action event of any transition, and call events that are not external *internal*. Initially the machines are in their respective initial local states, and the environment generates a subset of the external events and arbitrarily changes the inputs to the system, enabling transitions as described above. Different statechart-based languages disagree on which enabled transitions are taken and what effects the taken transitions produce. We adopt the semantics of RSML [14] and STATEMATE [11]: Two transitions are non-conflicting if they do not share the same source local state, and a *maximal* set of enabled transitions that are pairwise non-conflicting, collectively called a *microstep*, is simultaneously taken—the

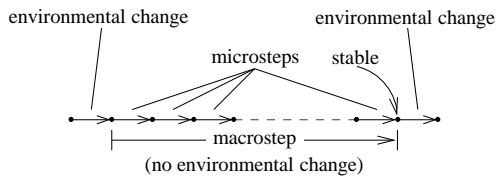


Figure 2: Microstep, macrostep, and synchrony hypothesis

system leaves the source local states of the transitions, enters the destination local states, and generates the action events (if any). The generated action events may trigger additional transitions in the next microstep. Events are instantaneous, so unless regenerated they disappear after the microstep.

In our example in Figure 1(a), event x is external. The guarding conditions c and d are assumed to be Boolean inputs, but they could have been predicates on other local machines not shown. Both machines are initially in *off*. When x occurs and c is true, the transition from *off* to *on* in machine A is enabled and taken, generating y . If d is true, then event y will in turn trigger the transition from *off* to *on* in machine B in the next microstep, and generates z .

The system is *stable* when no events occur. The sequence of microsteps between the time when the system is unstable and the time when it becomes stable again is called a *macrostep*. (A microstep and a macrostep are called a step and a superstep respectively in STATEMATE, whereas in RSML they are called a microstep and a step respectively.) The *synchrony hypothesis* says that during a macrostep no external events can arrive and the environmental inputs remain unchanged; that is, the system is infinitely faster than the environment [1]. Figure 2 depicts these notions. RSML enforces the synchrony hypothesis, while STATEMATE optionally allows it. We assume the synchrony hypothesis, which is central to the issues and techniques discussed in this paper.

2.2 Styles: Oblivious vs. Non-Oblivious

Figure 1(b) shows another way to specify machines A and B . Instead of generating event y to indicate local state change, machine A now generates y to signal its completion and pass the execution to machine B , which reacts based on the A 's local state. ($Prev(A)$ in the figure refers to the local state of A at the end of the previous macrostep.) Intuitively, we call such statecharts *oblivious* in the sense that the sequence of events generated and thus the synchronization are independent of the local states or inputs: In this case, for example, y is generated after x occurs regardless of the condition c and the local state of A ; likewise, z is generated after y regardless of B and d . In other words, a difference between the two systems arises when event x occurs but, say, machine A is *off* and c is false, in which case in Figure 1(a) no transitions are enabled and no internal events generated. In the same situation, y is still generated in Figure 1(b). Despite this difference, the stable-state behaviors of the two systems are identical.

A few observations are worth noting. In the non-oblivious system, the events are used for both synchronization (executing machine B after machine A) and local control (directing machine B 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 synchronize the machines in the desired order. While 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. This is consistent with Harel and Naamad's comment that in RSML a macrostep appears to be the “basic operation,” while in STATEMATE a microstep is the basic operation [11, p. 323]. Notice, however, that the differences arise not from the syntax or semantics, but from the distinct mental models of the system that the specifiers have.

3 SYMBOLIC MODEL CHECKING

We review the model-checking problem and the idea of symbolic model checking in this section.

3.1 The Model-Checking Problem

To analyze statecharts using state-exploration techniques, we view the system as a *global structure* $\langle Q, R, I \rangle$, 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 is 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 state space Q by declaring a set of state variables as follows. For each state machine m , declare a state variable whose range is the local states of m . (An additional state variable with the same range is used if $Prev(m)$ is needed.) For each event e , declare a Boolean state variable, which is true if and only if e occurs. For each input, 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 $\bigwedge_{m \in M} m = m_0 \wedge \bigwedge_{e \in E_i} \neg e$, 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, 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 [6]. It suffices here to point out that our encoding has the form $(\neg \text{stable} \rightarrow \text{micro}) \wedge (\text{stable} \rightarrow \text{env})$, where micro encodes microsteps, env encodes the environmental transitions across macrosteps, and stable indicates when the system is stable, namely $\neg \bigvee_{e \in E} e$ where E is the set of all events. Note that under this encoding a macrostep is represented as a sequence of global transitions. An alternative is to represent a macrostep as a global transition, but this would prevent us from analyzing behaviors within a macrostep, which is often useful for debugging purposes.

Many system properties can be expressed in the Computation Tree Logic (CTL) [8], a common temporal logic for model checking. 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 $X\phi$ means that ϕ holds on the path starting at the next state, $G\phi$ means that ϕ holds everywhere on the path, and $\phi W \psi$ means that ϕ holds everywhere before ψ holds, and ϕ must hold forever if ψ never holds. For example, the formula $AG\text{safe}$ asserts that the proposition safe holds in all reachable states, and $AG(\text{request} \rightarrow A(\text{request} W \text{response}))$ asserts that once issued, a request will persist unless a response is given.

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

3.2 Symbolic Search

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

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

Consider the formula AGp , which is true if and only if the proposition p holds in every reachable state. Let $P \subseteq Q$ be the set of states that satisfies p . As shown in Figure 3, we can evaluate the formula by performing a *backward* breadth-first search from the set Y_0 of states that immediately violate p to

```

 $Y_0 := Q - P; Y := Y_0; i := 0$ 
repeat
   $i := i + 1$ 
   $Y_i := Pre(Y_{i-1}) - Y$ 
   $Y := Y \cup Y_i$ 
until  $Y_i = \emptyset$ 

$AGp \text{ iff } Y \cap I = \emptyset.$


```

Figure 3: Backward search for AGp

find the set Y of states that may eventually violate p . Symmetrically, the formula can also be evaluated by performing a *forward* breadth-first search from the initial states I to find the set of reachable states. The loops, guaranteed to terminate for finite state spaces, are said to compute *fixed points*. In conventional “explicit” search, Y is implemented as a hash table, while the *search frontier* Y_i is implemented as a queue. More complicated temporal-logic formulas can be evaluated in similar ways by computing one or more fixed points [8].

The method is impractical for many large systems because of the sheer number of states that must be explored. More efficient for large state spaces are *symbolic searches* [9, 4]. A state set (e.g., Y) can be symbolically encoded as a predicate over the state variables, just as we encoded the initial states I in Section 3.1. The idea then is to manipulate this predicate directly to explore the whole set without enumerating its elements. Because we are dealing with finite state spaces, we can assume without loss of generality that each state variable is Boolean, so each such predicate is a Boolean function, which can be represented as reduced ordered binary decision diagrams (BDDs) [3]. Boolean operations, satisfiability checking, and predecessor computation 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.

4 SOME INTUITIONS ON BDD SIZE

In this section, we examine some factors that can affect the BDD size. We argue that oblivious systems tend to produce smaller BDDs than non-oblivious ones, and that backward searches may explore simultaneous local transitions that actually cannot be taken simultaneously, causing BDD blowup. Note that we focus on backward searches in this paper, because we have not found forward searches efficient in our experiments. One possible reason for the inefficiency is that forward searches from the initial states maintain all system invariants, whether or not they are relevant to the properties being checked. There may be many such invariants relating local states, events, and inputs in nontrivial ways, resulting in large BDDs [12].

4.1 Oblivious vs. Non-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, non-oblivious systems have many more ways to finish a macrostep than oblivious ones, and a backward search from the stable states needs to capture all these possibilities, producing larger BDDs.

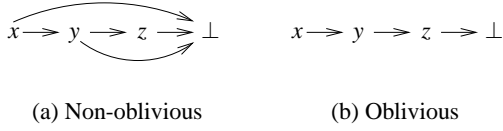


Figure 4: The precedence relation \prec

To elaborate, we define \prec to be a binary relation over the events E together with a special symbol \perp , which intuitively represent stable states. 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 . In addition, for each event e , we have $e \prec \perp$ if there exists a state q in which e occurs and q has a stable successor state.

Figure 4 shows the precedence relations for both the non-oblivious and oblivious systems in Figure 1. Note that there are fewer edges pointing to \perp in Figure 4(b); for example, the edge (x, \perp) is absent because x always triggers y and thus never immediately results in a stable state. The presence of fewer edges generally indicates fewer dependencies among the state variables, and tends to generate smaller BDDs. For example, assume that we want to check whether machine A can be in *on* in a stable state. The backward search can be illustrated in Figure 4 by traversing the graphs in reverse in a breadth-first manner. We start from \perp and follow the edges backward: For Figure 4(a), we visit states in which either x , y , or z occurs, while for Figure 4(b), we visit only states in which z occurs. The BDD in the former case is larger because of the additional constraints from x , y , and the state machines triggered by them.

To make the example more concrete, we encode the state space of the nonoblivious system with Boolean state variables x, y, z, a, b, c and d , where a (resp. b) is true if and only if machine A (resp. B) is in *on*. Refer to the backward search in Figure 3. We have $Y_0 \equiv \text{stable} \wedge a \equiv \neg(x \vee y \vee z) \wedge a$. What are the predecessors of Y_0 in Y_1 ? Such a state must be unstable, so one of the events must occur. In addition, in order for the successor state to be stable, none of the transitions should be enabled. So to be in *on* in the successor state, machine A must be in *on* already. With some derivations, it can be shown that we have

$$Y_1 \equiv a \wedge ((x \wedge c) \vee (y \wedge (b \leftrightarrow d)) \vee z),$$

quite complicated for even this simple example. Now consider the oblivious system. Unlike the situation above, a predecessor of Y_0 cannot have events x or y occurring, because otherwise y or z respectively will always be generated, leading to an unstable state. So we have $Y_1 \equiv \neg x \wedge \neg y \wedge z \wedge a$. In the subsequent iterations, we have $Y_2 \equiv \neg x \wedge y \wedge a$, and $Y_3 \equiv x \wedge c$, etc., all simple functions. In addition, unlike the non-oblivious case, b and d never appear in the functions and therefore the BDDs. This is appealing because the property being checked indeed does not depend on B and d . (We also need an extra Boolean variable to encode $\text{Prev}(A)$, but this too never appears in these BDDs.) In this example, the fixed

points computed are the same in both cases, but the BDDs generated are smaller for the oblivious system.

4.2 Unreachable Simultaneous Transitions

A source of BDD blowup common to both oblivious and non-oblivious systems is the exploration of many unreachable simultaneous transitions. For instance, a backward search does not know that, in both Figures 1(a) and 1(b), the transitions in machine A can never be enabled simultaneously with any transition in machine B in any reachable state, and therefore may explore such simultaneous transitions.

This problem was attacked in an earlier work by identifying a subset of events (such as x, y, z in our examples) that are mutually exclusive in all reachable states, and incorporating this fact into the transition relation to prune the search [7]. The BDD size was dramatically reduced as a result. However, for reasons that we will not elaborate, this method is more effective for oblivious than for non-oblivious systems. Another problem is that the method is not effective if most pairs of events may occur simultaneously (as is the case in our case study in Section 6).

5 OPTIMIZATION

In the previous section, we saw some reasons for large BDDs. Armed with those intuitions, we systematically modify the global structure to decouple the synchronization from the local control and to prune backward searches, and at the same time preserve most system properties: Often, the maximum length of a macrostep can be statically bounded. In this case, we can make every macrostep equal in length, and incorporate a *microstep counter* into the system. 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.

5.1 Microstep Counter

In what follows, we assume the precedence relation \prec to be *acyclic*, that is, $(e, e) \notin \prec^+$ for each e , where \prec^+ is the 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. (This assumption is not essential for our technique, but it makes bounding the maximum length of a macrostep easy, as we now show.)

For each event e , let $\lambda(e)$ be the smallest set of integers with

1. $\lambda(e) = \{1\}$ if e is an external event, and
2. $i \in \lambda(e) \Rightarrow i + 1 \in \lambda(e')$ for all e' with $e \prec e'$.

Intuitively, i is in $\lambda(e)$ if e can occur just before the i^{th} microstep of some macrostep. Since \prec is acyclic, the integers in $\lambda(e)$ are bounded, and the values of $\lambda(e)$ for all e can be computed in $O(|E|^3)$ total time by traversing the precedence graph. The maximum length k of a macrostep is then the largest integer in $\lambda(e)$ for any e . For Figure 1(a), we have

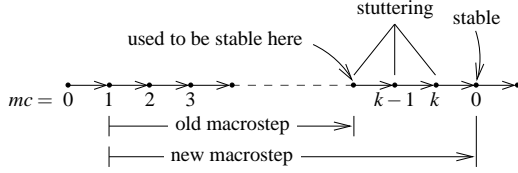


Figure 5: Effects of microstep counter on system behaviors

$\lambda(x) = \{1\}$, $\lambda(y) = \{2\}$, $\lambda(z) = \{3\}$, and $k = 3$. Note that some macrosteps may have fewer than k microsteps.

To symbolically encode a statecharts model as a global structure, in addition to the usual state variables, we define a microstep counter mc to range from 0 to k . The behavior of the microstep counter depends only on the set E_x of external events (the primed variables below encode the next state):

Modification 1 (Microstep 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 with

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

and conjoin the transition relation R with

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

Stability now depends only on the microstep counter:

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

The 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 zero. Whenever some external event occurs, mc becomes one in the same state and a macrostep begins. The value of mc is then incremented by one in every subsequent microstep until the value reaches k . At that point, it will be reset to zero in the successor states, and the system will be stable. Note that the internal events and the local states do not come into the picture, and that every macrostep has exactly k microsteps.

Clearly, the local transitions in the statecharts are unaffected by the changes, but the stable state may be delayed as illustrated in Figure 5—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 [13], and every CTL formula without the next-time X operator is preserved by stuttering [2]. (Formulas with the X operator can count the number of microsteps and thus may not be preserved.)

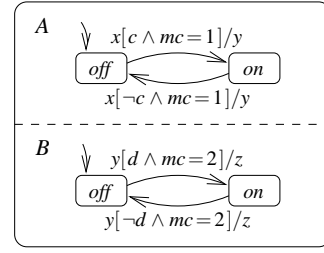


Figure 6: Guard the machine in Figure 1(a) with mc .

Our final modification uses the microstep counter to guard transitions.

Modification 3 (Guards). Each transition labeled with $e_1[cond]/e_2$ is encoded in the global structure as if it were labeled with $e_1[cond \wedge mc \in \lambda(e_1)]/e_2$.

One can intuitively think of the new label as

$$mc \in \lambda(e_1)[e_1 \wedge cond]/e_2.$$

In other words, the transition is triggered by the microstep counter, and the event e_1 becomes part of the logic of the guarding condition. Notice, however, that this modification cannot affect the system's behavior, because in any reachable state, the occurrence of e_1 implies $mc \in \lambda(e_1)$. This can be proved by induction on the definition of λ . So the inclusion of $mc \in \lambda(e_1)$ is redundant as far as forward behavior is concerned. We make the following claim:

Claim (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 zero by referencing *stable*).

To see how these modifications help, consider again the example in Section 4.1: We want to search backward in the non-oblivious system from $Y_0 \equiv \text{stable} \wedge a$. Figure 6 shows the modified machines with transitions guarded by the microstep counter. Notice that we no longer have $x \prec \perp$, because by the construction of mc , the external event x 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 $y \prec \perp$. So the relation \prec now resembles the one in Figure 4(b) instead of Figure 4(a). Specifically, Y_1 is now $mc = 3 \wedge a$, a lot simpler than before. In fact, in this example, the search looks similar to the one for the oblivious system without the microstep counter; in particular, b and d also never show up.

To see that the modifications help prune unreachable simultaneous transitions in backward searches, observe that the microstep counter in Figure 6 makes it explicit that the transitions in machine A and B are mutually exclusive. The technique here is more general than using mutual exclusion of

events mentioned in Section 4.2. For example, if $\lambda(e_1) = \{1, 2\}$ and $\lambda(e_2) = \{2\}$, 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 $mc = 2$).

Recall that our construction of the microstep counter makes certain macrosteps longer so as to make every macrostep equal in length. This generally results in an increased number of iterations to reach a fixed point, affecting the performance in a negative way. Nevertheless, in our case study reported in Section 6, this impact was 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.

5.2 Condition-Driven Transitions

Some variants of statecharts, such as STATEMATE, allow transitions not guarded by events. That is, a transition can have labels of the form $[cond]/acts$, where $cond$ is a guarding condition and $acts$ is a list of action events. Such transitions are enabled when the machine is in the source local state and $cond$ is true. We call these transitions *condition-driven*. Intuitively, instead of checking the guarding condition only when triggered by an event, condition-driven transitions continuously poll the guarding condition.

Extending our techniques to handle condition-driven transitions requires a more general framework, and we omit the details in this paper. 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 i^{th} microstep. For systems with every transition triggered by some event, the set $\rho(t)$ is simply $\lambda(e)$ 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.

6 CASE STUDY

The techniques presented in the previous section were motivated by the analysis of a statecharts model of the electrical power distribution (EPD) system on the Boeing 777 aircraft. We briefly describe the model, discuss some results of the analysis, and report the benefits of the optimization technique given in the previous section. We also discuss how model checking could potentially be used to benefit the model-based development processes used at Boeing. 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 [17].

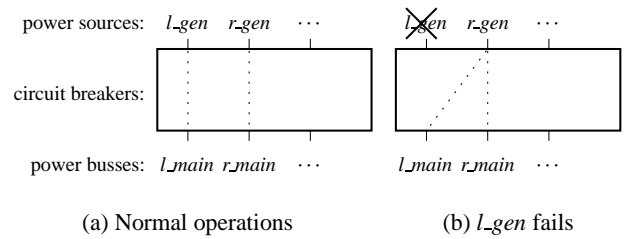


Figure 7: Handling a power-source failure in the EPD model

6.1 The EPD Model

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 busses via a number of relayed circuit breakers. Failures of the power sources or circuit breakers are automatically detected and isolated. We focus on the portion of the statecharts that models the main and backup AC distribution subsystems. There are 33 two-state machines, 23 Boolean inputs, and 34 events, for a total of 90 Boolean state variables, or about 10^{27} global states, of which at least 10^{15} are reachable.

Figure 7(a) depicts part of the system configuration in normal operations. The power busses 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 Figure 7(b). The same configuration may also result from failures in the circuit breakers that connect l_gen and l_main . The system is supposed to satisfy a number of stringent requirements, such as the resilience of the power busses against single or multiple failures in the power sources and/or the circuit breakers.

A circuit breaker, either open or closed at any moment, is modeled as a two-state machine and is managed by a controller. Figure 8 shows a generic circuit breaker and its controller. The transitions in the circuit-breaker state machine

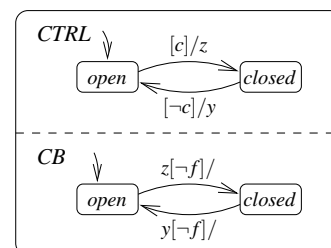


Figure 8: A circuit breaker (CB) and its controller (CTRL)

are guarded by the complement of a Boolean input f that indicates a failure, so a failed circuit breaker does not respond to the controller. The guarding condition c of the controller is usually a nontrivial predicate relating inputs and the local states of other circuit breakers and the power sources.

6.2 Analysis

We analyzed the main and backup AC power subsystems by translating the statecharts to the input language of the CTL model checker SMV [15]; other subsystems were abstracted away manually. The analysis can be divided into analysis on normal behaviors (i.e., no component failures) and fault tolerance (single and multiple failures). We report some of the more interesting results here. Although the model had been exercised extensively in simulation, several flaws were discovered using model checking. We were able to obtain these results only after using our optimization technique presented in Section 5.

Normal Operations

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

$$\text{AG}((\text{stable} \wedge \text{no-failures}) \rightarrow (\text{main} \wedge \text{backup})) \quad (1)$$

where *no-failures* is a proposition indicating the absence of failures, and *main* and *backup* assert respectively that the main busses (l_main and r_main) and backup busses 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 busses be powered when there are no failures, they should be powered by different sources. We checked the formula

$$\text{AG}((\text{stable} \wedge \text{no-failures}) \rightarrow \text{separate-sources}) \quad (2)$$

where the proposition *separate-sources* asserts that a power source is connected to at most one bus. This time, however, the model checker gave a counterexample revealing a bug in the model of the circuit breakers. In the counterexample, r_gen initially powers both l_main and r_main because of a failure in the circuit breakers. Now assume the failed circuit breaker is modeled by the machine CB in Figure 8. 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 busses, 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 busses should in fact tolerate one failure in the power sources or circuit breakers. We checked the formula

$$\text{AG}((\text{stable} \wedge \text{at-most-1-failure}) \rightarrow \text{main}) \quad (3)$$

where the proposition *at-most-1-failure* has the obvious meaning. The model checker gave a counterexample that again reveals the bug B1, although the scenario is more complex. It involves a failure in a circuit breaker, a change in inputs to induce a state change in its controller, the circuit breaker’s recovery, and a subsequent failure in one of the power sources. After we fixed the bug and rechecked the formula, the model checker gave another counterexample that discloses 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 busses. We refer this bug as B2. (We have not attempted to fix this bug in this study.)

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

$$\text{AG}((\text{stable} \wedge \text{at-most-2-failures}) \rightarrow \text{at-least-1-backup}). \quad (4)$$

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

Miscellaneous

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

$$\text{AG}(\text{main} \rightarrow \text{A}(\text{main} \text{W} \neg \text{no-failures})) \quad (5)$$

which asserts that, even in the unstable states, if the main busses 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 busses. 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 busses should be powered.

6.3 Performance

Despite the results we finally obtained, the initial experience of the analysis was daunting—the BDDs generated were enormous and the fixed-point computations could not go beyond two or three iterations before we ran out of memory. As a result, even trivial formulas could not be evaluated, let alone the formulas given above. We attacked the problem by focussing on a small part of the system, trying alternative ways of modeling, and looking for specific reasons for the BDD blowup. Hand-simulation of the symbolic search was sometimes used to build up intuition. The technique given in Section 5 results from the insights gained in the process.

Table 1 shows, for each formula, the number of search iterations, the time (in seconds), and the number of BDD nodes (in thousands) needed to compute the fixed point. (Formula 5 requires computing two fixed points to evaluate, and its number of iterations in the table is the sum of the two numbers.) All these searches were performed on the model without fixing the bugs B1 or B2. The results were obtained on a Sun Ultra-2 workstation using SMV version 2.4.4 (augmented with a conjunctive partitioning heuristic [18]). The data suggest dramatic improvements made by our optimization technique, without which the evaluation of each of the properties was not feasible. The rightmost column gives the data for computing the reachable states of the optimized model using a forward search, and shows the superiority of backward searches for our system.

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

6.4 Discussion

A major goal of the case study was to evaluate the use of model checking as a debugger in support of requirements validation at Boeing by providing an additional debugging tool over and above the existing use of simulation. The use of modeling and simulation to support requirements validation at Boeing is described in Nobe and Bingle [16]. 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

Table 1 Resources needed for computing fixed points when the optimization in Section 5 was used. (Without optimization, every formula could not be evaluated in two hours of CPU time.)

	formulas					forward search
	1	2	3	4	5	
# iteration	40	40	30	40	61	49
time (sec.)	85	54	49	462	89	3806
nodes (K)	837	402	464	2965	452	3865

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 focussing on the main AC subsystem and ignoring the backup AC subsystem.

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 experience enabled us to identify oblivious synchronization as a feature of statecharts that facilitates model checking—the decoupled synchronization and local control tend to make the BDDs representing state sets smaller when backward symbolic search is used. We devised an innovative technique of introducing a microstep counter into the model to achieve a similar effect in systems with non-oblivious synchronization, and to prune backward searches. The technique works by bounding the length of a macrostep and using a microstep counter to synchronize local transitions. The improvements were crucial for the case study, as they allowed analysis that used to be infeasible to complete in just several minutes of CPU time.

Our optimization technique aims 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 *dynamically* during the search [5, 12, 19]. They

are quite general and work for large classes of circuits. We implemented some of these techniques and applied them to the EPD model, but the results were not satisfactory. In contrast, the technique we developed concentrates on statecharts and *statically* changes the underlying global structure. Intuitively, the technique uses information from forward analysis on event precedence to realign the search frontiers and to prune backward searches. This strategy of combining forward syntactic analysis and backward searches appears to be a promising approach to improving the efficiency of symbolic model checking.

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

ACKNOWLEDGEMENTS

We thank Greg Taleck for his initial work on translating and analyzing the EPD model and Kurt Partridge for helpful comments on an earlier version of the paper.

REFERENCES

- [1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992. 3
- [2] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1/2):115–131, July 1988. 6
- [3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(6):677–691, August 1986. 1, 4
- [4] 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*, 98(2):142–170, June 1992. 1, 4
- [5] G. Cabodi, P. Camurati, and S. Quer. Improved reachability analysis of large finite state machines. In *1996 IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, pages 10–14, San Jose, California, USA, November 1996. 9
- [6] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998. 1, 1, 4
- [7] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In M. Young, editor, *ISSTA 98: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 102–112, Clearwater Beach, Florida, USA, March 1998. Published as *Software Engineering Notes*, 23(2). 1, 5
- [8] 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. 4, 4
- [9] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: International Workshop Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag. 4
- [10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 1, 2
- [11] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996. 2, 3
- [12] A. J. Hu and D. L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV'93 Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 3–14, Elounda, Greece, June/July 1993. Springer-Verlag. 4, 9
- [13] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Computer Congress*, pages 657–668, Paris, France, September 1983. North Holland. 2, 6
- [14] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9), September 1994. 1, 2
- [15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 8
- [16] C. R. Nobe and M. G. Bingle. Model-based development: Five processes used at Boeing. In *Proceedings of the IEEE International Conference and Workshop: Engineering of Computer-Based Systems*, Jerusalem, Israel, March/April 1998. 2, 9
- [17] C. R. Nobe and W. E. Warner. Lessons learned from a trial application of requirements modeling using statecharts. In *Proceedings of the 2nd International Conference on Requirements Engineering*, pages 86–93, Colorado Springs, USA, April 1996. IEEE. 2, 7
- [18] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *Proceedings of IEEE/ACM International Workshop on Logic Synthesis*, Lake Tahoe, USA, May 1995. 9
- [19] K. Ravi and F. Somenzi. High-density reachability analysis. In *1995 IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, pages 154–158, San Jose, California, USA, November 1995. 9
- [20] 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*, pages 77–88, Gaithersburg, Maryland, USA, June 1996. IEEE. 1