Model Checking Large Software Specifications

William Chan Richard J. Anderson Paul Beame Steve Burns Francesmary Modugno
David Notkin Jon D. Reese

Abstract—In this paper we present our experiences in using symbolic model checking to analyze a specification of a software system for aircraft collision avoidance. Symbolic model checking has been highly successful when applied to hardware systems. We are interested in whether model checking can be effectively applied to large software specifications. To investigate this, we translated a portion of the state-based system requirements specification of Traffic Alert and Collision Avoidance System II (TCAS II) into input to a model checker (SMV). We successfully used the model checker to analyze a number of properties of the system. We report on our experiences, describing our approach to translating the specification to the SMV language, explaining our methods for achieving acceptable performance, and giving a summary of the properties analyzed. Based on our experiences, we discuss the possibility of using model checking to aid specification development by iteratively applying the technique early in the development cycle. We consider the paper to be a data point for optimism about the potential for more widespread application of model checking to software systems.

Index Terms—Formal methods, state-based specifications, requirements, statecharts, symbolic model checking, binary decision diagrams, software verification.

1 Introduction

Errors in software specifications cost money and, in some cases, threaten lives [8, 43]. How can we increase our confidence in the specifications, particularly those of safety-critical systems? Formal methods offer opportunities for mechanical verification, but most existing techniques either do not scale to large systems, require extensive human guidance, or are limited to verifying simple (though important) properties like deadlock freedom, consistency, and completeness.

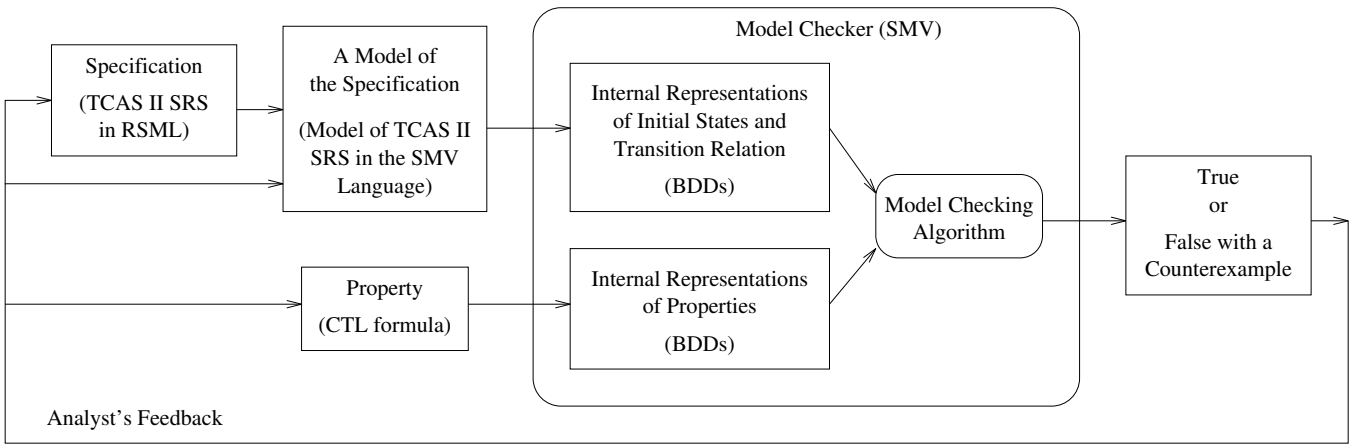
Symbolic model checking [15] based on binary decision diagrams (BDDs) [10] is an efficient automatic verification technique that is simultaneously capable of scaling and of verifying a wide range of properties (Section 2). It has been applied successfully to many industry-scale hardware circuits, but not aggressively to the analysis of software specifications. In this paper we describe an experience in analyzing a large system requirements specification using symbolic model checking.

In our experiment, we translated (Sections 3 and 4) a significant portion of a preliminary version of the Traffic Alert and Collision Avoidance System II (TCAS II) System Requirements Specification from the Requirements State Machine Language (RSML) [44] into input to the Symbolic Model Verifier (SMV) [45]. TCAS II is an aircraft collision avoidance system required on many commercial aircraft and has been described as "the most complex system to be incorporated into the avionics of commercial aircraft" [44, p. 685]. We were able to control the size of the BDDs representing the specification (Section 5) so that we could analyze a number of properties (Section 6). These include general robustness properties as well as some safety-critical properties specific to the domain.

Our objective was to test the effectiveness of model checking on software systems, so our experiences in applying the technology are more important than the individual results. One intent is to convey how we overcame some key obstacles, with the hope that most or all of these techniques are applicable to other situations. We stress two approaches that we found crucial in overcoming the complexity and size of the specification, making it more amenable to symbolic model checking: the use of nondeterministic modeling primarily to abstract nonlinear arithmetic and to allow checking part of the specification, and the use of an iterative process to analyze the specification. We discuss related work (Section 7), as well as point out some limitations of the current model-checking techniques and tools, and suggest some future research directions (Section 8).

Our analysis was based on preliminary versions of the specification, mainly on the version 6.00, dated March 1993. We

- *A version of this paper appeared in* IEEE Transaction on Software Engineering*, 24(7), July 1998, pp. 498–519.* *Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.*
- *W. Chan, R.J. Anderson, P. Beame, and D. Notkin are with the Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA. Email: {wchan, anderson, beame, notkin}@cs.washington.edu.*
- *S. Burns is with Strategic CAD Technologies, Intel Corporation, Jones Farm Campus, Hillsboro, OR 97124, USA. Email: smburns@ichips.intel.com.*
- *F. Modugno is with the University of Pittsburgh, PA 15260, USA. Email: fm@cs.cmu.edu.*
- *J.D. Reese is with Safeware Engineering Corporation, Seattle, WA 98102, USA. Email: jdreese@safeware-eng.com.*



did not have access to later versions, so we do not know if the properties identified here are present in later versions.

This article is a full-length report of the conference version of the paper [5].

2 Model Checking

Model checking is a formal-verification technique based on state exploration. Given a state transition system and a property, model checking algorithms exhaustively explore the state space to determine whether the system satisfies the property. Figure 1 is a schematic of the process of model-checking a state-based specification, with the instances that we used for the components shown in parentheses. A model of the specification and a property are fed to a model checker. The result is either a claim that the property is true or else a *counterexample* (a sequence of states from some initial state) falsifying the property. In practice, counterexamples often provide valuable debugging information, and can be used by the software engineer to modify the specification, the model, or the property checked. This iterative process is inherent in our work.

In the rest of this section, we give an overview of the basics of CTL model checking and SMV, the model checker that we used.

2.1 The CTL Model Checking Problem

In temporal-logic model checking, we are given a state transition system, which models a software or hardware system, and a property specified as a formula in a certain temporal logic, and determine whether the system satisfies the formula. A common logic for model checking is the branching-time Computation Tree Logic (CTL) [19], which extends propositional logic with certain temporal operators. Typical formulas include the following (meanings of the temporal operators such as AG will be given later):

AG *safe*: All reachable states are *safe*.

AG AF *stable*: The system is *stable* infinitely often.

AG (*request* → AF *response*): A *request* is always followed by a *response* sometime in the future.

AG EF *restart*: It is possible to *restart* the system in any reachable state.

Formally, a state transition system $\langle Q, R, I \rangle$ consists of a set of states Q, a state *transition relation* $R \subseteq Q \times Q$, and a set of initial states $I \subseteq Q$. A *path* is an infinite sequence of states such that each consecutive pair of states is in R. The set of states Q is often encoded by a set of *state variables*, such that each state corresponds to some valuation for the variables and no distinct states correspond to the same valuation (that is, the mapping of Q to the variable valuations is one-to-one).

For simplicity we discuss just a subset of CTL, namely the subset with only the temporal operators AG, AF, EG, and EF, which are sufficient to understand our examples. We can recursively define this restricted class of CTL formulas as follows: We say that a *proposition* is any Boolean combination of predicates on the state variables. A *formula* is either a proposition, a Boolean combination of formulas, or of the form AG f, AF f, EG f, or EF f, where f is a formula.

Each formula is evaluated at some state q. A proposition holds at q if q satisfies the proposition. The operator A means "for all paths starting at q", E means "for some path starting at q", G means "for every state along the path", and F means "for some state along the path". So AG *safe* holds at q if every state (G) along every path (A) starting at q satisfies the proposition *safe*.

The system satisfies a formula if the formula holds at all initial states. If not, a model checker typically attempts to find a counterexample. For example, if the formula AG *safe* is false, a counterexample is a finite path starting at some initial state and ending at a state that is not *safe*.

Readers familiar with temporal-logic model checking may notice that, although a CTL formula is usually interpreted over a Kripke structure in which every state is labeled by a set of atomic propositions, in our definition, a state is not explicitly labeled, but can be thought as being labeled implicitly by its corresponding state-variable valuations. This more restricted formulation is sufficient for our presentation.

2.2 Symbolic Model Checking and BDDs

In explicit model-checking techniques, the truth value of a CTL formula is determined in a graph-theoretic manner by traversing the state diagram, with time complexity linear in the size of the state space and in the length of the formula [19]. Unfortunately, the size of the state space is often exponential in the size of the system description, resulting in the *state explosion problem*.

An important breakthrough in model checking was the introduction of *symbolic* techniques: Instead of visiting individual states as in conventional state space search, symbolic model checkers visit a *set* of states at a time [15, 45]. A state set can be represented by a predicate on the state variables such that a state is in the set if and only if the predicate is true at the state. The efficiency of symbolic model checking relies on succinct representations and efficient manipulations of these predicates.

When the state space is finite, we can assume without loss of generality that the state variables are Boolean and there are only finitely many of them. A predicate on these variables is simply a Boolean function, which can be represented by reduced ordered binary decision diagrams (BDDs) [10]. Intuitively, a BDD is like a binary decision tree, except that isomorphic subtrees must be combined resulting in a directed acyclic graph. In addition, each path can contain a variable at most once, and must comply with a fixed linear order of the variables. BDDs are canonical (that is, given a Boolean function and a variable order, there exists a unique BDD that represents the function) and Boolean operations such as conjunction, disjunction, and negation can be computed in polynomial time. BDDs are usually small, but often their sizes depend critically on the variable order (as we will see in Section 5.3).

A number of BDD-based symbolic model checkers have been built, mainly for hardware circuit verification. They represent state sets, and often the transition relation, as BDDs. Because of the efficiency of BDDs and their algorithms, hardware systems with over 10^{20} states have been analyzed [16], and many industrial designs have been verified or falsified.

2.3 SMV

SMV [45] is a CTL symbolic model checker using BDDs to represent state sets and transition relations. Below we summarize the SMV features pertinent to our discussion. In SMV, `1` represents *true*, and `0`, *false*. The logical operators *and*, *or*, and *not* are `&`, `|`, and `!`, respectively.

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

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

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

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

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

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

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

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

A common way to define next-state values is to use a `case` expression:

```
ASSIGN
  next(x) := case
               x<7: x+1;
               1: 0;
             esac;
```

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

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

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

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

of *sym*. For example, `next(d)` is identical to `next(x)=7 & next(b)=0`.

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

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

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

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

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

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

A program can contain both next-state assignments and `TRANS` statements. Their conjunction forms the transition relation.

3 TRANSLATION BASICS

In this section, we give an informal overview of RSML, and provide intuition of the translation from RSML to SMV by showing an example. General translation rules will be described in the next section.

3.1 RSML Overview

RSML [44] is a state-machine language based on statecharts [31], extending conventional state diagrams with state hierarchies and broadcast communications. Focusing on a subset of RSML, we model a system by a state hierarchy, events, and inputs; in particular, the input and output interfaces in RSML are ignored.

State Hierarchy State hierarchies allow the machine to have deep and orthogonal structures. More precisely, each state S may contain *substates*, whose *superstate* is S. The state S is either an *and-state* or an *or-state*. Intuitively, the machine is in S if and only if (1) it is an and-state and the machine is in all of its substates, or (2) it is an or-state and the machine is in exactly one of its substates. Each or-state has exactly one *default* substate; intuitively, if the machine enters an or-state, it also enters its default substate unless it explicitly enters some other substate. If a state has no substates, it is an *atomic* state.

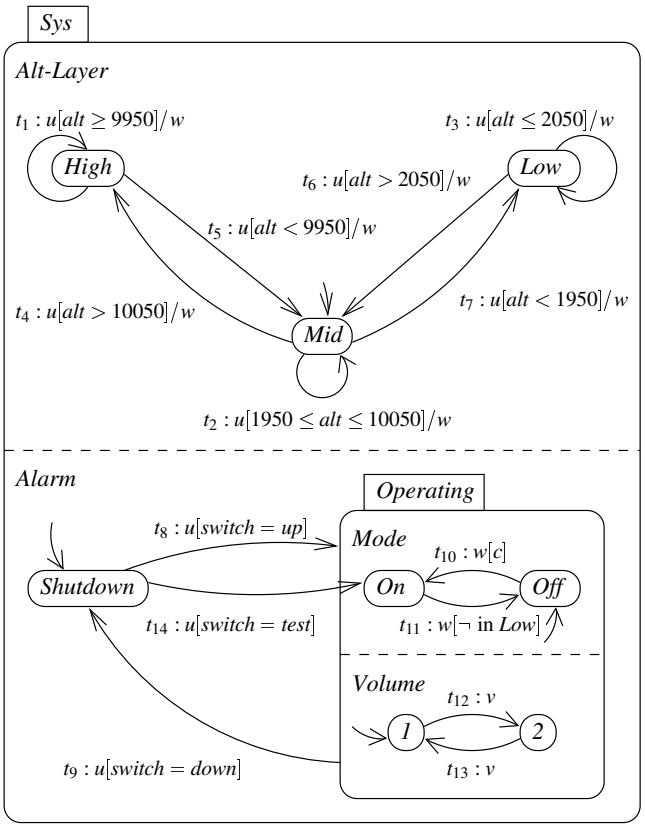


Figure 2: An example of an RSML machine

Figure 2 shows an example of an RSML machine. It only illustrates some of the features of RSML and statecharts, and does not represent any real device. It triggers an alarm when the altitude of an aircraft is too low according to certain criteria. The hierarchical structure is shown in the diagram by containment. At the highest level, *Sys* is an and-state, whose substates are *Alt-Layer* and *Alarm* (substates of an and-state are separated by dashed lines). *Alt-Layer* is an or-state with three substates, *High*, *Mid*, and *Low*. *Alarm* is also an or-state, with substates *Shutdown* and *Operating*; the latter is an and-state, containing *Mode* and *Volume*. Default states, e.g., *Mid*, are indicated by arrows without origins. Figure 3 shows the hierarchy as a tree.

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

States in RSML are synchronized by events, which are broadcast to the entire system. There are three events in the example: u, v, and w; the first two are generated by the environment and are called *external* events. The environment is supposed to generate u periodically, and v is generated when the pilot changes the volume of the alarm. The event w is generated by the machine for internal synchronization. For sim-

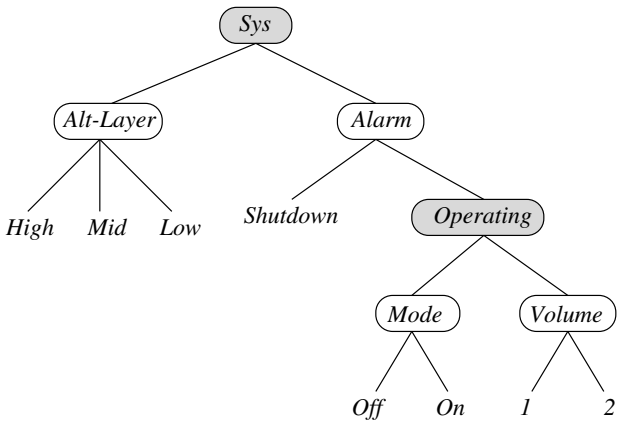


Figure 3: The state hierarchy drawn as a tree. The shaded nodes represent and-states, unshaded nodes represent or-states, and the leaves are atomic states.

Transition(s): *Off* $\longrightarrow$ *On*

Location: *Mode*

Trigger Event: *w*

Condition:

			OR	
	Alt-Layer **in state** Low	T	T	T
AND	$alt < 1000$	T	.	.
AND	$alt < 1500$	.	T	.
AND	$\text{PREV}(alt) < 1500$	.	T	.
AND	$t \geq t(\text{Exited}(Mid)) + 5$	.	.	T

Output Action:

Figure 4: Transition from *Off* to *On*

plicity, we assume in general that an event is either generated by the environment or by the RSML machine, but not both.

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

$$id\text{: } trig[cond]/acts$$

where *id* uniquely identifies a transition and is only used for our presentation; *trig* is a *trigger* event; the guarding condition *cond* is a predicate on states and inputs; and *acts* is a set of *action* events. The guarding condition and the actions are optional. The idea is that if the machine is in the source state, the trigger occurs, and the guarding condition is true (it is considered true if absent), then the transition is *enabled*. If no other *conflicting* transitions are enabled (intuitively two transitions conflict if they cannot be taken at the same time), then this transition is taken: The machine exits the source state, enters the target state, and generates the action events. Additional states may be entered or exited to maintain the integrity of the state hierarchy. For example, if t_{14} is taken, the state *On* is entered, so are *Operating*, *Mode*, *Volume*, and *1*.

A machine operates as follows. Initially, the environment generates some external events, enabling transitions as described above. A *maximal* set of enabled transitions that are mutually non-conflicting is then taken, possibly generating new events. This is called a *microstep*. (Notice that if there are conflicting transitions enabled, then this maximal set is not unique, resulting in nondeterminism.) After each microstep, all the events except those newly generated vanish. These new events are broadcast to the whole machine and may trigger other transitions. This process continues until no more transitions are enabled, at which point the machine becomes *stable*. This cascading of microsteps, from the point when the external events arrive to the point when the machine becomes stable, is called a *step*. RSML assumes the *synchrony hypothesis* [6], which says that during a step, no new external event may occur and the values of the inputs remain unchanged. In other words, the machine runs infinitely faster than the environment. Once the machine is stable, inputs can change and external events can again occur.

AND/OR Tables The guarding condition *c* of transition t_{10}, too complex to fit in Figure 2, is shown in Figure 4 as an *AND/OR table*, one of the features that distinguish RSML from statecharts. The leftmost column of the table shows a list of predicates, and the table represents a proposition over these predicates in disjunctive normal form. That is, each column (except the leftmost one) evaluates to the conjunction of the predicates marked T in that column (or their negations if any were marked F), and the entire table evaluates to true if one or more of its columns is true. Informally, the expression $\text{PREV}(expr)$ refers to the value of *expr* at the end of the previous step. The special variable t indicates the current time, while t(Exited(*S*)) is the time when state *S* was last exited. (Note that the synchrony hypothesis implies that the value of t does not change during a step.) So the table reads: (row 1) the machine is in state *Low*, and either (column 1) the current value of *alt* is less than 1000, or (column 2) both the current and previous values of *alt* are less than 1500, or (column 3) the machine exited *Mid* at least 5 time units ago.

To reduce the sizes of the AND/OR tables, RSML allows *functions* and *macros*. For example in Figure 4, instead of an input, *alt* could have been a function defined elsewhere in the requirements, and its value might depend on inputs and states. Similarly, a macro can replace a primitive predicate in the leftmost column, and is defined as an AND/OR table elsewhere. Functions and macros can optionally take parameters.

3.2 Translating the Example

In this subsection, we translate the RSML example above to SMV code. The complete SMV program is shown in Appendix A.

SMV Variables First, we declare the SMV variables for the state hierarchy, inputs, and events. The events are easy—they

are naturally translated to Boolean variables. For example, we have:

```
VAR
  u: boolean;
```

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

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

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

```
alt: 0..20000;
```

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

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

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

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

because the machine is always in *Sys*, and thus always in *Alt-Layer* and *Alarm* as well. We also have

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

Conditions for other states can be defined similarly.

RSML Transitions Now we can define when a transition, say t_7, is enabled:

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

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

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

We also need to specify the event generated:

```
next(w) := t1|t2|t3|t4|t5|t6|t7;
```

and initialize the states and event:

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

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

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

```
DEFINE
  stable := !(u|v|w);
```

Then the input *alt* changes according to the following assignment.

```
ASSIGN
  next(alt) :=
    case
      stable & !next(stable): 0..20000;
      1: alt;
    esac;
```

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

```
next(u) :=
  case
    stable: {0,1};
    1: 0;
  esac;
```

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

PREV and Timing Constraints We have already translated most of the RSML machine except two predicates in the AND/OR table in Figure 4. Referencing the previous value of *alt* requires the introduction of an extra variable `prev-alt` to remember its value at the end of a step:

```
VAR
```

```
  prev-alt: 0..20000;
ASSIGN
  next(prev-alt) :=
    case
      stable: alt;
      1: prev-alt;
    esac;
```

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

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

The timer `time-Mid` indicates the number of time units passed since *Mid* was last exited. The timer is reset when *Mid* is exited via transitions t_2, t_4, or t_7. At the end of a step, the value of the timer is incremented unless it is already 5—since we only care whether the timer is at least 5, specific values greater than 5 are irrelevant.

Note that taking t_2 is viewed as exiting and re-entering *Mid*. This is consistent with the semantics of RSML. However, if t_2 were specified as a so-called *identity transition*, then taking the transition would not reset the timer. In that case, we would simply leave out `t2` in the first case branch. For simplicity, we will not further discuss identity transitions in this paper.

4 Translation Rules

To explain the translation from RSML to SMV more generally and precisely, we first formally define an RSML machine as a state transition system given in Section 2.1, based on the operational semantics of RSML by Leveson et al. [44]. Some of our definitions are based on Pnueli and Shalev [49]. For simplicity, we first assume the absence of timing constraints and PREV functions. Then we show how we translate deterministic RSML machines and certain nondeterministic machines to SMV programs. Timing constraints and PREV functions are considered later in this section, along with some discussions on alternative semantics and translation rules.

4.1 RSML Machines as State Transition Systems

We define an RSML machine as a state transition system $\langle Q, R, I \rangle$. To distinguish between an RSML state and an element in Q, we call the latter a *global state* and call R the *global transition relation*.

RSML States Let *States* be the finite set of RSML states, and let $Children\colon States \to \mathcal{P}(States)$ map each state to its substates, or children. The function is required to impose a tree structure on the states with a distinguished element *root* as the root of the tree. We sometimes use *parent* as synonym for *superstate*.

We define $Children^+$ and $Children^*$, the transitive and reflexive-transitive closures of *Children*, as

$$Children^+ = \bigcup_{i \geq 1} Children^i$$
$$Children^* = \bigcup_{i \geq 0} Children^i,$$

where for each state p in *States*,

$$Children^0(p) = \{p\}$$
$$Children^{i+1}(p) = \bigcup_{s \in Children(p)} Children^i(s) \quad \text{for } i \geq 0.$$

If $s \in Children^*(p)$, then we say that s is a *descendant* of p, that p is an *ancestor* of s, and that s and p are *ancestrally related*. If in addition $s \neq p$ (that is, $s \in Children^+(p)$), then s is a *strict descendant* of p, and p is a *strict ancestor* of s.

If $Children(s) = \emptyset$, then s is an atomic state. Otherwise, it is either an and-state or an or-state; in the latter case, it has exactly one default substate.

Intuitively, a *configuration* is a maximal set of states that the machine can be in simultaneously. That is, $C \subseteq States$ is defined to be a configuration if

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

For example, in Figure 2, the states {*Sys*, *Alt-Layer*, *High*, *Alarm*, *Shutdown*} is a configuration.

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

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

Let $External \subseteq Events$ be the set of external events. The set I of initial global states is the set of every triple (C, E, V) with $C = Complete(root)$, $E \subseteq External$, and $V \in Inputs$.

RSML Transitions Let *Trans* be the set of RSML transitions. Each transition $tr \in Trans$ has five attributes: the source state $src(tr) \in States$, the destination state $dest(tr) \in States$, the trigger event $trig(tr) \in Events$, the guarding condition $cond(tr) \subseteq \mathcal{P}(States) \times Inputs$, and the action events $acts(tr) \subseteq Events - External$.

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

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

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

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

Formally, for any transition *tr*, we define $Exits(tr)$ as $Children^*(maxsrc(tr))$ and $Enters(tr)$ as the intersection of $Complete(dest(tr))$ and $Children^*(maxdest(tr))$. $Enters(tr)$ precisely specifies the states that the machine enters on taking transition *tr*. $Exits(tr)$ is a little less precise. Clearly, before transition *tr* is taken, the machine is in some states in $Exits(tr)$, in particular $src(tr)$, and after *tr* is taken the machine is no longer in any state in $Exits(tr)$. The mere fact that *tr* is taken does not in general specify any more information than this about the states that the machine is in prior to the transition. As an example, $Exits(t_{14})$ is $\{Shutdown\}$, $Enters(t_{14})$ is $\{Operating, Mode, On, Volume, 1\}$, $Exits(t_9)$ is all the descendants of *Operating*, and $Enters(t_9)$ is $\{Shutdown\}$.

The global transition relation

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

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

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

(b) No two transitions in T conflict.

(c) T is maximal: Every transition not in T but enabled in (C,E,V) conflicts with some transition in T.

(d) $C' = (C - \bigcup_{tr \in T} Exits(tr)) \cup \bigcup_{tr \in T} Enters(tr)$.

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

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

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

4.2 Translate Global States

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

The optimal encoding for this or-state is obviously to declare one variable with a range of size n (or equivalently declare $\lceil \log n \rceil$ Boolean variables). The encoding described in Section 3.2 is a natural extension of this idea. Recall that for each or-state s, we declare a variable with range $Children(s)$. To obtain a more succinct encoding, we also flatten nested or-states, i.e., or-states whose superstates or substates are also or-states. For example, taken from the TCAS II requirements, Figure 5 shows an example of nested or-states—Composite-RA, RA, and Positive are all or-states. (The vertical bar on the right together with the arrows attached to it is a *transition bus*, implying a transition between every pair of states connected to the bus. That is, No-RA, Climb, Descend, and Negative are pairwise connected by a transition in either direction.) Our translated SMV program contains the following code:

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

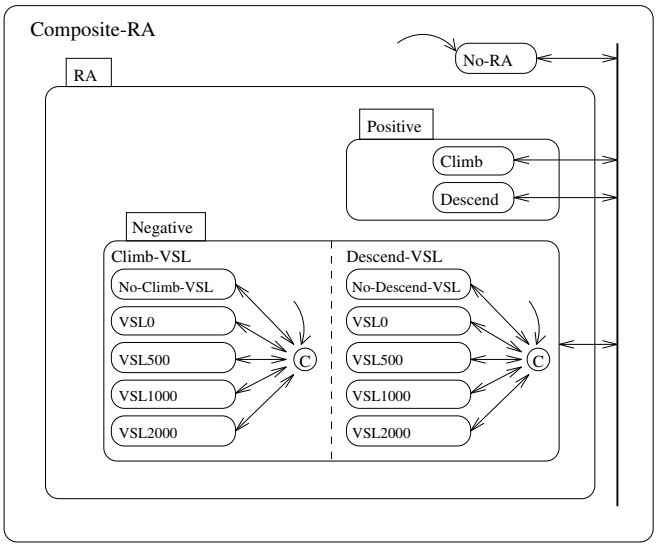


Figure 5: Nested or-states in TCAS II

```
in-Positive := in-Climb | in-Descend;
in-Climb := in-Composite-RA
              & Composite-RA = Climb;
```

More generally, let O be the set of or-states whose parents (if any) are and-states, and let $\mathcal{A}$ be the set of and-states or atomic states that have or-state parents. That is, the set $O \cup \mathcal{A} \subseteq States$ consists of states at the upper boundaries of the alternations between or-states and and-states in the state hierarchy. Note that *root* is not contained in $\mathcal{A}$, but may or may not be in O. For each $s \in \mathcal{A}$, its leader, denoted as $leader(s)$, is its lowest ancestor in O; and for each $p \in O$, the set of its followers, denoted as $Followers(p)$, consists of every $s \in \mathcal{A}$ whose leader is p. Note that this leader-followers relationship is identical to the parent-child relationship, when there are no nested or-states and nested and-states, and every atomic state has an or-state as parent, as is the case in our example in Figure 2. In Figure 5, the followers of Composite-RA are No-RA, Climb, Descend, and Negative, while those of Climb-VSL are its children.

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

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

1. *For each $p \in O$:*
   `VAR`
   p`: Followers(`p`);`
   `ASSIGN`
   `init(`p`) :=` $default^*(p)$`;`
2. `DEFINE`
   `in-`*root* `:= 1;`
3. *For each $s \in \mathcal{A}$:*
   `DEFINE`
   `in-`s `:= in-`$leader(s)$ `&` $leader(s) = s$`;`
4. *For each and-state or atomic state $s \notin \mathcal{A}$ with parent p:*
   `DEFINE`
   `in-`s `:= in-`p`;`
5. *For each $s \in O$ with parent p:*
   `DEFINE`
   `in-`s `:= in-`p`;`
6. *For each or-state $p \notin O$:*
   `DEFINE`
   `in-`p `:=` $\bigvee_{s \in Children(p)}$ `in-`s`;`
7. *For each $e \in Events$:*
   `VAR`
   e`: boolean;`
8. *For each $e \in Events - External$:*
   `ASSIGN`
   `init(`e`) := 0;`
9. *For each input variable y:*
   `VAR`
   y`:` $Range(y)$`;`

Figure 6: Rules for declaring and initializing SMV variables for RSML machines

4.3 Translate Deterministic Transitions

Deterministic machines are easy to translate because the set of transitions taken is exactly the set of enabled transitions. Figure 7 shows translation rules that are correct only for deterministic machines. If these rules are applied to a nondeterministic machine, the behavior of the translated SMV program will be identical to the RSML machine up to the point when some conflicting transitions are simultaneously enabled; after that, the two systems will start to exhibit diverging behaviors.

Rule 10 defines when a transition is enabled and taken. (In Section 3 and Appendix A, for simplicity, we define one symbol *tr* instead of defining *tr*`-enabled` and *tr*`-taken`.) In the rule, $cond(tr)$ refers to the proposition that describes the guarding condition.

Rule 11 defines the effects of a transition on the state hierarchy. The set $Followers(p) \cap Enters(tr)$ contains the follower of p that the machine enters upon taking *tr*. Note that this set is either empty or a singleton set.

Rules 12–15 generate the appropriate events and update the inputs, during and at the end of a step.

10. *For each tr ∈ Trans:*
```
DEFINE
 tr-enabled := in-src(tr) & trig(tr) & cond(tr);
 tr-taken := tr-enabled;
```
11. *For each p ∈ O:*
```
ASSIGN
 next(p) :=
  case
   • For each tr ∈ Trans with
       Followers(p) ∩ Enters(tr) ≠ ∅,
     let s be the unique state in the set:
    tr-taken: s;
   • For the default branch:
    1: p;
  esac;
```
12. *For each e ∈ Events − External:*
```
ASSIGN
 next(e) := ⋁_{tr: e∈acts(tr)} tr-taken;
```
13. *For each e ∈ External:*
```
ASSIGN
 next(e) :=
  case
   stable: {0,1};
   1: 0;
  esac;
```
14. *For each input variable y:*
```
ASSIGN
 next(y) :=
  case
   stable & !next(stable): Range(y);
   1: y;
  esac;
```
15.
```
DEFINE
 stable := ¬ ⋁_{e∈Events} e;
```

Figure 7: Rules for translating deterministic RSML transitions

We argue informally for the correctness of the translation. Clearly, the rules ensure that only enabled transitions can cause state change or generate action events. We claim that every enabled transition causes the necessary state change: In Rule 11, because of the deterministic assumption, at most one non-default case branch can be true, so each enabled transition *tr* always results in updating every variable that has a follower in $Enters(tr)$. By virtue of our state encoding, this implies that every state in $Enters(tr)$ is entered and every state in $Exits(tr)$ previously occupied is exited. Finally, it is easy to see that enabled transitions always result in the generation of their action events, and that inputs are updated correctly.

Note that the symbol `stable`, which indicates the end of a step, is defined as $\neg \bigvee_{e \in Events} e$, but a direct translation from the definition of R would be $\neg \bigvee_{tr \in Trans} tr\texttt{-enabled}$. The two versions are nearly identical, because if no events are occurring, there will be no enabled transitions, and if there are no enabled transitions, no events will occur in the next microstep. Defining `stable` using events is more concise, as there are usually far fewer events than transitions. A second advantage is that the BDD representing `stable` becomes much smaller. (More efficiency issues will be discussed in Section 5.)

10′. *For each tr ∈ Trans:*
```
DEFINE
tr-enabled := in-src(tr) & trig(tr) & cond(tr);
tr-taken := tr-enabled
          & ⋀_{s∈Enters(tr)} next(in-s)
          & ⋀_{e∈acts(tr)} next(e);
```
11′. *For each p ∈ O:*
```
TRANS
 (p = next(p)) ∨
 ⋁_{Followers(p)∩Exits(tr)≠∅} tr-taken
```
16′. *For each tr ∈ Trans:*
```
TRANS
 ¬tr-enabled ∨
 (¬tr-taken ↔ ⋁_{tr′∈Conflict(tr)} tr′-taken)
```

Figure 8: Rules for translating a class of nondeterministic RSML transitions

The example in Section 3 was translated mostly based on these rules. Careful readers may notice that the machine is actually nondeterministic, so the translation is not exact. In Section 6.1, we will discuss how to discover the violating transitions.

4.4 Translate Nondeterministic Transitions

In principle, translating machines with arbitrary nondeterministic transitions is straightforward. One strategy is to declare a set of auxiliary Boolean variables representing the transitions in *Trans*. We can translate the definition of the global transition relation R literally as a first-order logic formula over the finitely many auxiliary and global state variables, and then optionally quantify out the auxiliary variables. This conceptually simple method is inefficient, because the number of transitions, and thus the number of auxiliary variables, is usually large. There are other potentially more efficient ways of constructing the global transition relation, but in general the constructions are still expensive. Interested readers are referred to the work of Helbig and Kelb [35] for an example.

We now give modifications to the rules in Figure 7 to handle a rich class of nondeterministic machines, namely those with the property that a transition is taken if and only if it *appears* to be taken. Rule 10′ in Figure 8, replacing Rule 10, explains more precisely what this assumption means: A transition is taken if and only if it is enabled and in the next microstep the machine enters the appropriate states and generates the appropriate action events. We will see shortly why this may not be true in general.

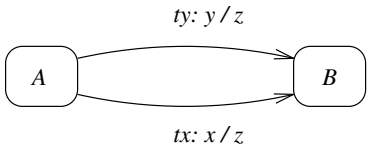


Figure 9: Transitions with same Sources and destinations

Rules $11'$ and $16'$ replace Rule 11. Rule $11'$ ensures that the machine remains in a state unless some transition exiting that state is taken. The set *Conflict*(*tr*) is defined as the transitions that conflict with *tr*. Rule $16'$ concisely says that the set of transitions taken must be maximal and nonconflicting: Either a transition is not enabled (in which case it cannot be taken), or it is not taken because one of the conflicting transitions is, or it is taken, in which case none of the conflicting transitions is. These rules are correct, if the definition of *tr*-taken is correct to begin with.

Figure 9 shows why the latter is not always true. If events x and y can occur simultaneously when the machine is in state A, then the two conflicting transitions are enabled. According to the semantics, the machine will take exactly one of *tx* and *ty*, go to state B, and generate event z. However, if we just look at the state change and the event generated, the machine will appear to have taken both *tx* and *ty*, making our definition of *tr*-taken incorrect. In fact, in this case, our translation prevents the machine from entering state B, because otherwise, Figure $10'$ would make both *tx*-taken and *ty*-taken true, which is precluded by Figure $16'$. (On the other hand, Figure $16'$ also prevents the machine from staying at state A, resulting in deadlock.)

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

Simpler Nondeterminism Although the class of machines captured above is quite rich and the translation does not introduce auxiliary variables, the resulting SMV program can be large because Rule $16'$ produces code with size quadratic in the number of transitions in the worst case. Furthermore, defining the transitions with the TRANS construct in SMV is more error-prone than using ASSIGN.

However, certain nondeterminism is easy to model. As an example, suppose we want to specify the state *Alt-Layer* in Figure 2 as an entirely nondeterministic machine: All the guarding conditions on t_1 through t_7 are omitted. The translation to SMV in this case is trivial:

```
ASSIGN
  next(Alt-Layer) :=
    case
      u: {High, Mid, Low};
      1: Alt-Layer;
    esac;
  next(w) := u;
```

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

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

We used assignments similar to these in our model of the TCAS II machine and they proved to be sufficient for our experiments.

4.5 Translate Timing Constraints

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

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

```
VAR
  θ: 0..k_θ;
ASSIGN
  next(θ) :=
    case
      ⋁_{tr∈T_θ} tr: 0;
      stable & θ < k_θ: θ+1;
      1: θ;
    esac;
```

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

checker search for an initial value that leads to violation of the property being checked.

Comparing two times, like Entered(s_1) > Exited(s_2), can also be handled by introducing extra variables. Although we have been assuming the discrete-time model (i.e., time is a natural number), it is possible to extend model checking to handle the dense-time model (i.e., time is a nonnegative real number), when we restrict to the same class of time predicates [2]. However, when t, Entered(s), or Exited(s) are used in arbitrary arithmetic expressions, whether discrete time or dense time is used, the machine cannot be precisely modeled as a finite-state system, and in fact, the model checking problem becomes undecidable [3].

4.6 Translate PREV

When the value of PREV(y) for some input y is needed, we use the following code:

```
VAR
  prev-y: Range(y);
ASSIGN
  next(prev-y) :=
    case
      stable: y;
      1: prev-y;
    esac;
```

Again, we do not initialize `prev-y` for the same reason that we do not initialize timers. The translation can be easily modified if y is a state, a macro, or a function.

An alternative translation for PREV(y) is to remember the truth values of the predicates involving PREV(y) instead of the value of PREV(y) itself. For example, for the predicate PREV(alt) < 1500 in Figure 4, we could remember the truth value of $alt < 1500$ instead of the numeric value of alt. This method has the advantage of possibly using fewer BDD variables, but is less general. For example, it cannot deal with predicates involving both previous and current values, like PREV(alt) < alt.

4.7 Miscellaneous

Other RSML constructs We have not exhausted all RSML constructs, but the rest are easy to translate: Macros and functions without arguments can be translated simply as defined symbols. Those with arguments can be translated as SMV *modules*, which are analogous to templates and can be instantiated at each call site of the macros or functions. RSML *state-machine arrays* give a succinct representation for isomorphic substates of an and-state. They can be translated to SMV as an array of module instances. We also have not detailed the translation of conditional connectives (© in Figure 5), which, roughly speaking, factor out common triggers or guarding conditions of a set of transitions. The conceptually simplest translation is to remove a conditional connective by conjoining each pair of incoming and outgoing transitions, although more efficient translations are possible.

Granularity of Global Transitions Note that when we defined the global transition relation in Section 4.1, we implicitly assumed that a global transition represents a microstep, which seems a natural choice.

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

Yet another possibility is to represent a microstep as a series of global transitions, which directly corresponds to the semantics of RSML given by Leveson et al. [44]. There, instead of being defined as a maximal set T of nonconflicting transitions, a microstep is equivalently defined by a loop: T is initially empty, and enabled transitions are added to T one at a time until a maximal set is obtained. One may therefore choose to represent each iteration in this loop as a global transition. An obvious drawback is the increased number of global transitions required to encode a microstep. However, a more serious problem is the introduction of *asynchrony* into the model: Even if T is unique (that is, the microstep is deterministic), there are in general many different orders of picking the transitions in T, and the model checking algorithm will need to explore all these possibilities. Our representation of microsteps can be viewed as a way of statically eliminating such asynchrony.

Alternative Semantics Some other variants of statecharts can be translated with similar rules. For example, the STATEMATE semantics [32] of statecharts is close to the semantics considered here. A notable exception is that the former does not insist on the synchrony hypothesis but provides it as an option. We can easily forsake the synchrony hypothesis by changing Rule 15 in Figure 7 to set `stable` to 1. STATEMATE also provides internal variables and allows assignments to them as actions. In addition, certain transitions that are considered conflicting here are assigned different priorities and thus do not result in nondeterminism when simultaneously enabled. Trigger events are also optional. Slight modifications to the rules would suffice for these differences. Other constructs like *history connectors* and synchronization through *activities* would require new translation rules.

In contrast, the semantics defined by Pnueli and Shalev [49] are quite different. It is unclear how to translate from their semantics in a simple way without introducing many auxiliary global state variables.

The RSML semantics defined by Heimdahl and Leveson [34] are slightly different from the semantics consid-

ered here, which are based the earlier work of Leveson et al. [44]. The differences become important when conflicting transitions with different triggers are simultaneously enabled, which does not happen in the portion of the TCAS II requirements machine that we modeled. In general, however, different translation rules would be required.

5 OBSTACLES

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

5.1 TCAS II

TCAS II is an airborne collision avoidance system required by the United States Federal Aviation Administration (FAA) on most commercial aircraft that enter U.S. airspace. The TCAS-equipped aircraft is surrounded by a protected volume of airspace. When another aircraft intrudes into this volume, TCAS II generates warnings (traffic advisories) and suggests possible escape maneuvers (resolution advisories, or RAs) in the vertical direction to the pilot. Examples of RAs include Climb, Descend, Increase-Climb ("increase the current climb rate"), Increase-Descend, Climb-VSL0 ("do not descend"), and Climb-VSL500 ("do not descend more than 500 ft/min").

The system requirements specification of TCAS II, a 400-page document, was written in RSML. The first obstacle to analysis was its sheer size. As a first attempt we decided to try to verify a portion of it, namely a state machine called Own-Aircraft, which occupies about 30% of the specification. Own-Aircraft has close interactions with another state machine called Other-Aircraft, which tracks the state of other aircraft in the vicinity and possibly generates RAs. Up to thirty other aircraft can be tracked. From the RAs given by all the instances of Other-Aircraft, Own-Aircraft derives a composite RA and generates visual and audio outputs to the pilot. The state shown in Figure 5 represents this chosen composite RA and is one of the substates of Own-Aircraft.

Since most of Own-Aircraft is supposed to be deterministic, we modeled it mainly based on the translation rules in Section 4.3, with the abstraction discussed below. We also created variables for any states of Other-Aircraft that are referenced within Own-Aircraft, and allowed nondeterministic transitions among the states using the translation explained in the last part of Section 4.4. We focused on resolution maneuvers with one intruder aircraft and thus modeled only one instance of Other-Aircraft.

5.2 BDDs

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

Numeric inputs are referenced in guarding conditions, macros, or functions. Although BDDs under a suitable variable order can efficiently represent equality and inequality between linear expressions (e.g., $2alt_1 + 3alt_2 > alt_3$), there is provably no efficient BDD representation for multiplication or division of variables (e.g., $alt \times time > distance$) under any variable order [11, 54]. So, we needed to avoid them. Two functions in Own-Aircraft do involve multiplication and division of values for measured altitudes and altitude rates. These are measurements of input variables that we already modeled nondeterministically. So we made the abstraction to treat the calculated values as nondeterministic themselves. (We also eliminated from our model several input variables that are only referenced by the two functions.) The abstraction did not cause problems for the properties that we checked and report in Section 6.

5.3 SMV

BDD Size and Linear Arithmetic The performance of BDD-based algorithms is directly related to the BDD size. Some of our early attempts at checking generated enormous BDDs: At one point the BDDs consumed 200 MB of physical memory, and other runs were terminated before the BDD was constructed. Our attempts to check formulas with the large BDDs were generally unsuccessful or too slow (our initial success in identifying nondeterminism discussed in Section 6.1 was an overnight run, which has been reduced to a few minutes).

The BDD size can be reduced by dynamic variable reordering and conjunctive partitioning [15], which are supported by the version of SMV that we used (Release 2.4.4). These techniques dramatically improved the performance of checking some formulas; however, they did not solve all the problems. The BDD size was very sensitive to the ranges of the variables representing altitudes and altitude rates. In fact, SMV cannot even efficiently handle our simple example program in Appendix A.

Initially we got around the problem by redefining the constants and reducing the variables to small ranges, for example, from 0 to 15 for altitudes and -4 to 3 for altitude rates. (Increasing the variables by one bit sometimes exploded the checking time from ten minutes to more than ten hours.) Although we were able to build the BDDs in this way and check some formulas, this *ad hoc* solution was unsatisfactory in many ways. An obvious drawback is that because of the small ranges, some distinct constants in the specification became identical after the mapping (for example, both 400 feet and 1000 feet might become 1). This changed the behaviors of the model and caused invalid analysis results.

We could not leave the results of addition and comparison nondeterministic as we did with multiplication and division in Section 5.2, because addition and comparison are essential to the logic of Own-Aircraft. For example, any Descend RA is prohibited when the difference between the current altitude of the own aircraft and the estimated ground level altitude is less than some threshold. If the subtraction or the comparison were modeled nondeterministically, this safety requirement would be violated trivially.

The problem with the ranges was due to SMV's inefficient implementation rather than the limitations of BDDs. As Yang et al. [56] observe, SMV is extremely inefficient in constructing BDDs for integers: Building the BDD for even a simple assignment like

```
next(x) := x;
```

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

```
VAR
  X: 0..N;
  Y: 0..N;
```

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

We considered two ways of attacking this problem, namely changing the internals of SMV, or doing addition and comparison at the source code level. Although in principle the former may be a better long term solution (Yang et al. [56] give an efficient algorithm for constructing BDDs for linear predicates), the latter method seemed a simpler approach and we were able to use it with great success. We wrote some simple `awk` scripts to automatically generate the code

```
VAR
  x1: boolean;   y1: boolean;
    ⋮              ⋮
  xn: boolean;   yn: boolean;
```

to declare X and Y, and `x1=y1 & x2=y2 & ... & xn=yn` to represent the equality $X = Y$. Addition, subtraction, and inequality can be similarly translated. We can now model the altitudes and altitude rates with the precisions required by the specification. Changing the variables for altitudes from 4 bits to 15 bits and those for altitude rates from 3 bits to 13 bits blows up the size of the state space roughly from 10^{40} to 10^{65}. However, this increase in precision increased the run time and the number of BDD nodes used by less than a factor of three. We also wrote an `awk` back-end to SMV to convert the bits back to integers for easy interpretation of the counterexamples.

Properties	Result	Time (sec)	No. of BDD nodes	Memory allocated (MB)
Building the transition relation	N/A	46.6	124618	7.1
Transition consistency	False	387.0	717275	16.4
Function consistency	False	289.5	387167	11.5
Step termination	True	57.2	142937	7.4
Descend inhibition	True	166.8	429983	11.8
Increase-descend inhibition	False	193.7	282694	9.9
Output agreement	False	325.6	376716	11.6

Table 1: Resources used in analysis

Counterexample Search Counterexamples also presented performance problems. Generating a counterexample often took hours even though the formula was determined false within minutes. Evaluating the formula and finding a counterexample were done by the model checker as two separate searches in the reachability graph. For example, to verify an invariant AG *safe* (that is, every reachable state is *safe*), the model checker started from the set of unsafe states, and iteratively searched backward to find the set of states that could reach some unsafe state. If this set contained any initial state, the model checker would determine the formula false and start a second, forward search from such an initial state to find a counterexample. We have modified the model checker by storing certain state information during the first search, eliminating most of the work in the second search. As a result, once such a formula is evaluated false, a counterexample can now be found almost instantly. The changes we made to the model checker are detailed elsewhere [18].

6 Results of Analysis

Once we overcame these obstacles, we were ready to do some analysis of the specification using the model checker. The properties that we analyzed include general properties that should hold in most RSML specifications (Sections 6.1–6.3) and domain-specific properties (Sections 6.4 and 6.5). The violation of some of the properties was unknown to us before the analysis (Sections 6.2 and 6.5).

We note that given an arbitrary system, it is often not obvious what domain-specific properties to verify. In our experiments, we based these properties on published documents and our own knowledge of the system. In this section, we only report the most interesting results we found. We will discuss some approaches to identifying properties to check in Section 8.4.

Table 1 shows the resources needed to analyze the properties. The time, the number of BDD nodes, and the mem-

ory allocated were reported by SMV. These include the resources used to construct the global transition relation, evaluate the formula, and find a counterexample if the formula was evaluated false. The first row gives the resources used just to build the global transition relation. The experiments were performed on a lightly loaded Sun SPARCstation 10 running SunOS 4.1.3 with 128 MB of main memory. We modeled the global state space with 227 Boolean variables, 10 of which are for events, 36 for the states of Own-Aircraft, 19 for the states of Other-Aircraft, 134 for altitude and altitude rates, 22 for inputs other than altitude and altitude rates, and 6 for other purposes. The size of the state space is about 1.4×10^{65}. The size of the *reachable* state space is at least 9.6×10^{56}. We obtained this lower bound by executing SMV with the command line option `-f` but without running it to completion. This option forces SMV to find the reachable state space before evaluating any formula.

6.1 Transition Consistency

We need to distinguish two kinds of nondeterministic transitions, namely those that are intentional, resulting from either the logic of the original specification (the rare case) or the abstraction that we employed (Section 5), and those that exist in the original specification but are unintentional, which we want to detect.

There are two reasons why we want to find the latter transitions. First, as Jaffe et al. [41] argue, nondeterminism in software requirements usually reflects *inconsistency* and should be avoided. Second, in our translation into SMV, we assumed the transitions are deterministic and separately dealt with the nondeterministic transitions as special cases (Section 4.4). If unintentional nondeterministic transitions are present, the SMV program in general will behave differently and all analyses will become invalid.

There are known nondeterministic transitions in early versions of the specification. For example, TCAS II has the notion of sensitivity level, which determines the volume of protected airspace around the aircraft. Some of the nondeterministic transitions allow a choice, under identical conditions, of increasing or decreasing the sensitivity level, which is clearly an inconsistency in these early versions of the specification. So, our first attempt was to find such transitions with the model checker. (For the other properties that we checked, we worked with a later draft specification, in which there are no inconsistencies in Own-Aircraft.) These nondeterministic transitions had previously been identified by Heimdahl and Leveson [34] using a different technique. We were interested in checking these properties to show that model checking could match previous results. In Section 7.3 we will summarize the differences between our model checking approach and the technique used by Heimdahl and Leveson.

In our example in Figure 2, transitions t_9 and t_{12} can be enabled simultaneously. We can check this with the model checker by the CTL formula

```
AG !(t9 & t12)
```

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

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

However, we can prove that this is not a problem. We say that a global state q is *reachable* if q appears on some path starting at some initial state, and a set of global states is reachable if *some* element in the set is reachable. We have the following lemma.

Lemma 1 *Given two state transition systems $M_1 = \langle Q, R_1, I \rangle$ and $M_2 = \langle Q, R_2, I \rangle$ with identical state spaces and initial states. Define*

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

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

The set N in the lemma is the set of "bad" global states that may lead to different behaviors in M_1 and M_2. The lemma says that some state in N is reachable in M_1 if and only if some state in N is reachable in M_2, but does not require that the two states be the same. Intuitively, this is true because the shortest path to any state of N in M_1 must also appear in M_2 and vice versa. A proof is given in Appendix B.

Let M_{RSML} and M_{SMV} be the state transition systems representing the RSML machine and the translated SMV program respectively, and let N be as defined above with M_1 and M_2 being M_{RSML} and M_{SMV}. Let's assume for now that abstraction by nondeterminism discussed in Section 5 was not used. By definition, the set N contains precisely the set of global states that are not faithfully translated. Because our translation handles deterministic transitions (and the intentional nondeterministic transitions) faithfully, the set N is exactly the set of global states that contains unintentional nondeterministic transitions. This means that N is reachable in M_{RSML} if and only if M_{RSML} exhibits unintentional nondeterministic behavior. Therefore, by the lemma, it is sufficient to analyze M_{SMV}

to detect the nondeterminism, and, in addition, we will not obtain false negative results. If, on the other hand, some intentionally nondeterministic transitions of M_{RSML} are mistakenly modeled as deterministic ones in M_{SMV}, this analysis will reveal them and the designer can then use this information to correct the model.

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

6.2 Function Consistency

The value of the function Displayed-Model-Goal, shown in Figure 10, is displayed to the pilot when an event called Composite-RA-Evaluated-Event occurs. (Most of the identifiers in the figure are RSML macros or abbreviations, the definitions of which are omitted here due to limited space.) The function represents the optimal altitude rate at which the pilot should aim (a positive value indicates the upward direction). The function definition consists of eight cases, which are supposed to be mutually exclusive. It is not obvious whether this is the case since the mutual exclusion depends on logic elsewhere in the specification.

Checking for mutual exclusion of the cases, which we call function consistency, is similar to checking for transition consistency in the previous subsection. We defined a Boolean symbol `Case-`i for the ith case, and checked the CTL formula

```
AG (Composite-RA-Evaluated-Event ->
    !((Case-1 & Case-2) |
      (Case-1 & Case-3) |
              ⋮
      (Case-6 & Case-7)))
```

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

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

6.3 Step Termination

A step in an RSML state machine may not terminate if the machine contains a cycle of events under the transition relation [44]. However, the precedence relation of the events in an RSML specification usually forms a partial order, so it is easy to see that a step will always terminate; this happens in the TCAS II specification. Alternatively, in our framework we can verify termination with the CTL formula

```
AG AF stable
```

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

6.4 Inhibition of Resolution Advisories

A TCAS II document [28] claims that (1) all Descend RAs are inhibited when the own aircraft is below 1000 feet above ground level, and (2) all Increase-Descend RAs are inhibited below 1450 feet above ground level. The logic that guarantees these safety properties resides in both Own-Aircraft and Other-Aircraft. We imposed the necessary constraints on the transitions of Other-Aircraft in order to check whether the part of the logic in Own-Aircraft is correct. The model checker found that while the first property is satisfied, the second is not. The formula that we checked for the second property was roughly

```
AG ((stable
     & Radio-Altimeter-Status = Valid
     & Own-Alt-Radio <= 1450)
    -> !Increase-Descend)
```

where Own-Alt-Radio is an input representing the altitude of the own aircraft above ground level, Radio-Altimeter-Status an input indicating whether Own-Alt-Radio is valid, and Increase-Descend an expression evaluating to true when an Increase-Descend RA is issued. (As mentioned in Section 5.3, the inequality is actually a long expression relating the bits of Own-Alt-Radio.) The counterexample given by the model checker revealed a typographical error in a guarding condition in the specification ($>$ instead of $\leq$).[1] The effect of the error was that the Increase-Descend RA was inhibited for only one step, thus allowing the safety property to be violated.

6.5 Output Agreement

In addition to the value of Displayed-Model-Goal, the state of Composite-RA in Figure 5 is also shown to the pi-

[1] We discovered the typographical error by observation during the translation process.

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

lot when Composite-RA-Evaluated-Event occurs. Therefore it seems safety-critical that Composite-RA and Displayed-Model-Goal agree with each other. We checked for several such properties. For example, one would expect that if Composite-RA is in state Climb, then Displayed-Model-Goal should be at least 1500 ft/min. However, the model checker revealed that this is not true. In fact, it showed a stronger result: When Composite-RA is Climb, Displayed-Model-Goal could be negative. The CTL formula we checked was the following:

```
AG ((Composite-RA = Climb
     & Composite-RA-Evaluated-Event)
    -> Displayed-Model-Goal >= 1500)
```

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

1. At time t_0, there is an intruder aircraft and Other-Aircraft gives a Descend RA. As a result, Composite-RA is in state Descend and by case 3 in Figure 10, we have Displayed-Model-Goal ≤ -1500 ft/min.

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

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

To the best of our knowledge, this behavior of the version of the specification was not known before. The resources shown in the last row of Table 1 are for this analysis.

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

```
AG ((Composite-RA-Evaluated-Event
     & New-Increase-Climb) ->
    Displayed-Model-Goal >=
      prev-Displayed-Model-Goal)
```

6.6 Miscellaneous

The value of any $\textsc{Prev}(expr)$ is undefined in the first step. As mentioned in Section 4.6, we did not constrain the initial value of the SMV variable representing $\textsc{Prev}(expr)$ to let the model checker find an initial value that falsifies the property being checked. So while verifying the properties mentioned above,

we also discovered situations in which PREV values are referenced in the first step.

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

```
AG EF Radio-Altimeter-Status = Valid
AG EF Radio-Altimeter-Status = Invalid
```

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

Another common use of AG EF formulas is to specify that it is always possible to shut down or restart the machine. While the notions of shutdown or restart are not applicable to our model of Own-Aircraft, we could check, for example, that it is always possible for the system to enter certain states or produce certain outputs (for example, the machine is never locked in a certain RA that no inputs can change). However, because nondeterminism was used to abstract out certain details, a behavior that is possible in the SMV program is not guaranteed to exist in the RSML machine. So the analysis of such AG EF formulas is not sound. (This problem can be solved by a recent technique called *module checking* [42].)

7 RELATED WORK

7.1 Case Studies

There have been several other independent case studies of model checking for real-life software requirements. In general, a major difference from our work is that their system environments were abstracted as a set of predicates or variables with a small enumerative range, whereas the inputs to our system include numerical values. Numerical calculation and comparison are abundant in the TCAS II specification, and they caused significant problems in the model checking process. These studies also differ from ours in the requirements languages used. For example, they do not contain features such as hierarchical states or microsteps, or do not assume the synchrony hypothesis.

Sreemani and Atlee [53] used SMV to analyze the A-7E aircraft software requirements, written in the Software Cost Reduction (SCR) notation [1, 36]. They successfully verified and falsified several temporal properties. From an informal specification of a hydroelectric power plant, Pugliese and Tronci [50] developed a process-algebra specification, which was then verified with an in-house BDD-based model checker. Crow and Di Vito [24] analyzed the requirements for a software subsystem on the Space Shuttle of NASA, using the explicit model checker Murφ [26] to verify invariants. Since symbolic representations like BDDs were not used, they manually reduced the ranges of the environment inputs to control the size of the state space.

Helbig and Kelb [35] gave a BDD encoding for statecharts. The version of statecharts semantics and the state hierarchy encoding that they used are similar to ours, except that they did not assume the synchrony hypothesis and did not flatten nested-or states (Section 4.2). With a custom-built BDD-based model checker, they encoded the transition relation more generally than we did to allow for arbitrary nondeterministic transitions, at the expense of the construction cost of the BDDs. We, in contrast, focused on deterministic transitions and certain nondeterministic ones that are easy to model and sufficient for our experiments. Their scheme was used to analyze a hypothetical production cell [25]. A compositional approach was used to cope with the BDD explosion problem.

7.2 Approaches to Fighting State Explosion

There are a number of other widely researched approaches to handling the state space explosion problem. In contrast to our work, which studies a single data point for a single approach, Corbett recently compared three approaches, BDD-based model checking (using SMV), partial-order state-space reduction, and inequality necessary conditions, all in the context of detecting deadlock in Ada tasking programs [23]. For deadlock, Corbett observed that "no technique was clearly superior to the others, but rather each excelled on certain kinds of programs." [23, p. 179]

Although the model of synchronization that Corbett considered was different from ours, some issues in the translation to SMV are relevant. He considered two translation schemes. One translation, on which he based his comparison with other techniques, precluded maximum parallelism among transitions (that is, some simultaneously enabled non-conflicting transitions are not allowed to be taken simultaneously and have to be taken in sequence), and therefore might not be optimal for symbolic model checking. The other translation, which he found less successful, did allow maximum parallelism, but used many extra variables. We, on the other hand, require maximum parallelism owing to the semantics of a microstep, and take pains to avoid introducing extra variables. Using a simple modification of our translation described in Section 4 for the class of problems he considered would only require extra variables where parallel nondeterministic transitions occur between the same two states. Use of such a translation might change the outcome of Corbett's comparison, but further work is needed to determine which approaches are most effective for checking particular properties on specific classes of systems.

7.3 Consistency and Completeness

Instead of exploring the state space, Heimdahl and Leveson compose results of local analysis to deduce global properties of the TCAS II specification [34]. However, the properties that we checked were different. Their concerns were transition consistency and completeness [41], which are domain-independent properties. In Section 6.1 we discussed how

we verified transition consistency. Completeness intuitively means that a response is specified for every input, and in principle can also be checked in our framework. In general, our approach permits analysis of arbitrary CTL formulas, and is therefore capable of verifying domain-specific properties as well.

Consider consistency in more detail. Their tool checks that the conjunction of the guarding conditions of every pair of conflicting transitions with the *same trigger* is a contradiction. That is, for the example in Figure 2, while we check whether the CTL formula `AG !(t1 & t5)` holds in the system, they check whether the conjunction of the guarding conditions of t_1 and t_5 is a contradiction. In general, their method can be less accurate, for three orthogonal reasons.

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

The second source of inaccuracy stems from their decision to consider only transitions with the same trigger. Consider again Figure 2. Conflicting transitions t_9 and t_{12} can be simultaneously enabled because their triggers u and v may occur at the same time. Their tool, however, would fail to detect them, simply because it never considers transitions with different triggers together. (On the other hand, if it makes the conservative assumption that any subset of events may occur at the same time, it will mistakenly report that t_9 and t_{10} may cause nondeterminism, without realizing that their triggers u and w are mutually exclusive in any reachable states.)

The last source of inaccuracy in their method is the way they construct a Boolean formula for checking: They create one Boolean variable for each predicate in the guarding condition. For example, to check whether t_4 and t_7 are mutually exclusive, they would have a Boolean variable x_1 for $alt > 10050$ and another variable x_2 for $alt < 1950$, and check whether $x_1 \wedge x_2$ is a contradiction. This clearly results in a false negative. Heimdahl and Czerny [33] use the theorem prover PVS [48] to attack this problem. On the other hand, for the same property, we would have 15 Boolean variables representing the bit encoding of *alt* and then construct the BDD for the predicate $alt > 10050 \wedge alt < 1950$, which is automatically reduced to a contradiction. Note that although we would have more Boolean variables, the BDD size scales well for inequalities and linear arithmetic operations. A disadvantage is the inability to deal with real numbers, which have to be discretized as bounded integers. BDDs also cannot efficiently handle the complicated nonlinear predicates in TCAS II, but currently neither can their theorem-proving approach [33].

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

7.4 Hybrid Systems

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

Verification of *hybrid systems* tackles this problem by modeling the environment with a set of real-valued variables governed by constraints on their derivatives [3]. The complexity of model checking becomes much higher and some problems even become undecidable, but symbolic model checkers for hybrid systems have been built [38]. Currently, they cannot cope with TCAS II: They cannot handle multiplication, and the sizes of the models analyzed in published case studies are orders of magnitude smaller than TCAS II. It would be interesting to see whether the next-generation tools can scale to significantly larger systems.

8 DISCUSSION

In this section, we address several common concerns about applying model checking to software, suggest the use of model checking as a development tool, and discuss some research directions.

8.1 Feasibility

The belief that model checking cannot apply well to complex software systems has been prevalent. This work (and other related work in Section 7.1) has shown why several concerns may not be as serious as commonly believed.

Restriction to Finite States One concern is that BDD-based model checking can only apply to finite state systems, but software is often specified with infinite states. A current research trend is to devise symbolic representations and model-checking algorithms to directly verify some classes of infinite state systems [3, 9, 14], although these techniques are far less mature than BDD-based methods. However, many infinite state systems can be abstracted as finite state ones, which are then amenable to conventional model-checking analysis [39, 55]. Often, the abstraction is conservative in the sense that, if the properties hold in the abstraction, they are guaranteed to hold in the full specification. If the goal of analysis is to find errors instead of proving correctness, this preservation guarantee can be forsaken, using techniques like model checking to find counterexamples but not to guarantee prop-

erties [40]. In our work, for example, some inputs of one of the versions of TCAS II are specified as real numbers, which were discretized as integers in our model (Section 5.2). The counterexamples we found in the finite-state model also exist in the full specification.

Regularity Another concern is that, unlike hardware circuits, software systems may not exhibit the necessary regularity to yield to symbolic techniques. On the contrary, we found that BDDs seem to capture the complex control structures of TCAS II. However, some "regular" operations, such as multiplication, do appear in the specification and cannot be handled efficiently by BDDs. Indeed, the data paths seem to be the real obstacle to analyzing the entire TCAS II. We will say more about arithmetic later.

Scale Our work demonstrates that symbolic model checking can be successfully applied to a real-life system that is widely recognized as complex. We have yet to analyze the entire specification, but this just shows that we can obtain useful results from incomplete models. In other words, it is not necessary to check a complete specification to get significant benefits from the technique.

Abstraction is the key to scale. In our experiment, most details in Other-Aircraft and arithmetic operations that are inefficient for BDDs were manually abstracted away by nondeterminism. Some form of abstraction can be automated by performing dependency analysis [7, 18].

Another approach to scale is automatic reduction techniques, of which the most relevant is perhaps BDD-based symmetry reduction [20]. The technique is not applicable to our current model of TCAS II because it lacks symmetries, but it will become a perfect candidate if we extend the model to include several instances of Other-Aircraft. (Recall that our current model contains only one instance of Other-Aircraft.) Symmetries arise in this case because two global states are equivalent if one can be obtained from the other by permuting the local states of the instances of Other-Aircraft.

8.2 Model Checking as a Design Tool

Understanding and Documentation As shown in Section 6.2, we sometimes obtained false counterexamples because of the over-abstracted model. Only when we refined the model to remove the spurious transitions could we verify the properties in question. This process of getting incorrect counterexamples and then removing them may seem counterproductive, but there are a number of reasons why this approach is in fact useful. A software engineer can use the information obtained from analyzing the counterexamples to clarify the relationship between parts of the specification, in particular between those parts that are fully modeled and those that are partially modeled. In complex specifications like TCAS II, the interconnections between the subsystems are often not fully described and documented. Our style of model checking can be viewed as a way of learning about and documenting the interconnections between parts of the specification.

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

Using verification techniques early in the development cycle to interleave design and analysis can tackle these problems. The complexity only gradually increases as the specification evolves, and verification at early stages is more likely to be tractable. In addition, analysis results can give fast feedback to designers to improve the cost-effectiveness of the technique. Researchers on hardware verification have also pointed out some advantages of early use of verification [46].

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

As an alternative to regression checking, if the abstracted model of Other-Aircraft *simulates* (in the sense of Milner [47]) a refined model, then every CTL formula in Sections 6.1–6.5 that holds in the former is guaranteed to hold in the latter as well. More generally, simulations preserve ACTL formulas, which intuitively include the CTL formulas without the E operator and with negations applied only to propositions. Efficient algorithms for simulation exist [22, 37] and provide an attractive way of hierarchical development of systems, although more experimental work for software specification is needed.

8.3 Tool Integration

Few, if any, integrated CASE tools today offer symbolic model checking as an option for verification. Software engineers who wish to use this technology may follow our strategy by translating their specifications to inputs to one of the model checkers available. Automatic translation is possible, as the rules in Section 4 show, although initially manual translation may help understand the subtleties that may arise in the translation process. Whether manual or automatic translation is used, our experiences showed that checking simple properties like transition consistency and completeness is useful in catching translation bugs.

Symbolic model checking, in its purest form at least, is conceptually simple and, with one of the available BDD pack-

ages [52], incorporating model checking algorithms into an integrated CASE tool should not be difficult in principle (although integration is almost always more difficult than anticipated [29]). The advantages of so doing compared with the translation approach include more flexibility in the construction of the BDDs, and more efficient and application-sensitive model checking algorithms. For example, the performance of the analysis in this paper can be improved by orders of magnitude by modifying the model checker [18].

It is important to design tools that domain experts feel comfortable in using. For example, AND/OR tables (Section 3.1) were designed to replace propositional logic for specifying guarding conditions in RSML because aircraft engineers did not find the latter natural [34]. Similarly, domain experts may not like temporal logic or understand its intricacies. Finding intuitive alternatives (perhaps even sacrificing some of the expressive power of the logic) is critical for gaining wider acceptance. Recently, Dwyer et al. have worked along this line and suggested using "specification patterns" [27].

8.4 Properties to Check

Model checking (or any form of property verification) is of no use if we do not know what properties to check. Finding a set of properties with good coverage can also increase our confidence in the correctness of the specification. A number of approaches can be used to address this complex question.

- The specification may already state some properties that are supposed to hold.
- Jaffe et al. [41] described a number of properties that should be satisfied by specifications (at least those with a safety-critical component): determinism, completeness, etc. We also believe that certain domain-specific properties like output agreement (Section 6.5) are applicable across many applications.
- Some other software analysis problems, such as deviation analysis [51], can be posed as model-checking problems.
- Some properties to check may arise in the field. For example, pilots have reported anomalous behavior that they observed while using some versions of TCAS II. Such anomalies could be checked against the specification, and one may determine if the problem is in the specification, in the implementation, or in the report itself.

8.5 Nonlinear Arithmetic

The most serious hurdle to applying BDD-based model checking to the remaining portion of the TCAS II specification is the abundance of the nonlinear arithmetic operations like multiplication of variables. It is not hard to see that model checking in the presence of predicates with integer multiplication is an intractable problem, at least as hard as factorization: Given a number z, we can build a transition with a guarding condition $xy = z \wedge x > 1 \wedge y > 1$, and determine whether z is a prime number by model-checking whether the transition is never enabled; finding a counterexample corresponds to factoring z. (Bryant and Chen observed a similar connection between BDDs and factoring [12].) Nevertheless, we should note that multiplication does not change the worst-case complexity of model checking, which is already a theoretically intractable problem without it. It follows that any symbolic technique is a heuristic. Theoretically, BDDs require exponential size for almost all Boolean functions; the efficiency of BDDs is due to their ability to capture concisely many of the control and data patterns that arise in practice. Similarly, there may be other techniques or representations that can handle the nonlinear arithmetic calculations that arise in practice.

It is tempting to adapt BDDs to handle multiplication. Word-level model checking is one such technique [21]. There, control is represented using BDDs and operations on integers are represented using *binary moment diagrams* [13], which can concisely represent the product of two integers. The algorithm in Clarke et al. [21] allows multiplication in the temporal logic formula, but not multiplicative predicates in the guarding conditions. It is unclear how this method can be adapted to solve our problem.

Some of us have proposed tightly coupling BDDs with a decision procedure for nonlinear predicates to attack the problem [17], but more work is needed to investigate its practicality. Another possible approach is approximation, as a middle ground between abstracting out multiplication completely and representing it precisely.

8.6 More Case Studies

Additional experience is needed in applying model checking to realistic state-based specifications. We are in the early stages of studying a statecharts specification of an electrical distribution system for avionics. But many additional experiments are needed to determine the overall applicability of model checking to state-based software specifications. As mentioned above, we also feel strongly that the most effective use of this technology will come in aiding the development of specifications, rather than in the after-the-fact checking of them. The real benefit of this approach can only be shown in practice by developing specifications with model checking.

9 Conclusion

We have shown how to translate part of a large system requirements specification into input to a symbolic model checker, and check several non-trivial properties. Our approach to analyzing the specification iteratively, by modeling some components nondeterministically and then refining them, proved to be powerful. These are critical steps towards realizing symbolic model checking as an effective tool in the process of analyzing and developing software specifications.

We believe that this investigation contributes to an increase in optimism that symbolic model checking can overcome predicted impediments and thus be successful in the analysis of realistic software specifications.

A TRANSLATION EXAMPLE

This is the complete SMV translation of the RSML example in Figure 2 (but as explained in Section 5.3, SMV does not handle this program well as it is).

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

```
next(prev-alt) :=
  case
    stable: alt;
    1     : prev-alt;
  esac;
next(time-Mid) :=
  case
    t2|t4|t7              : 0;
    stable & time-Mid < 5: time-Mid + 1;
    1                     : time-Mid;
  esac;
```

B Proof of Lemma 1

We assume that N is reachable in M_2 and argue that it is reachable in M_1 as well. The other direction is symmetric.

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

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

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

$$q_j \notin N \text{ for every } j < k. \tag{2}$$

We argue q_k is reachable in M_1 by showing that $q_0, q_1, \ldots, q_k$ is a prefix of some path in M_1. Since by assumption $q_0 \in I$, we only need to show $(q_j, q_{j+1}) \in R_1$ for every $j < k$.

For every such j, from (1) we know that $(q_j, q_{j+1}) \in R_2$. If $(q_j, q_{j+1}) \notin R_1$, then by the definition of N, we will have $q_j \in N$, contradicting (2). Therefore, we must have $(q_j, q_{j+1}) \in R_1$. □

Acknowledgments

We wish to thank the other members of the Winter 1996 CSE 590MC seminar at the University of Washington. This work was performed while all the authors were with the University of Washington, and was funded, in part, by National Science Foundation grant CCR-970670. William Chan's work was funded in part by a Microsoft-endowed graduate fellowship.

References

[1] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software requirements for the A-7E aircraft. Technical report, Naval Research Laboratory, March 1988.

[2] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Proceedings: 5th Annual Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, USA, June 1990. IEEE.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger, N. Halbwachs, P.-H. Ho, X. Nicollini, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[4] R. Alur and T. A. Henzinger, editors. *Computer Aided Verification, 8th International Conference, CAV'96 Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, USA, July/August 1996. Springer-Verlag.

[5] 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, USA, October 1996. Also published as *Software Engineering Notes*, 21(6), November 1996.

[6] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[7] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proceedings of the 1st ACM SIGPLAN Workshop on Automatic Analysis of Software*, Paris, France, January 1997.

[8] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[9] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger [4], pages 1–12.

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

[11] R. E. Bryant. On the complexity of VLSI implementations and graph representation of boolean functions with applications to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[12] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, June 1994.

[13] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *32nd ACM/IEEE Design Automation Conference, Proceedings 1995*, pages 535–541, San Francisco, USA, June 1995. ACM/IEEE.

[14] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state programs using Presburger artihmetic. In Grumberg [30], pages 400–411.

[15] 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*, 13(4):401–424, April 1994.

[16] 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.

[17] 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 Grumberg [30], pages 316–327.

[18] 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).

[19] 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.

[20] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, August 1996.

[21] E. M. Clarke, M. Khaira, and X. Zhao. Word level model checking—avoiding the Pentium FDIV error. In *33rd Design Automation Conference, Proceedings 1996*, pages 645–648, Las Vegas, USA, June 1996. ACM/IEEE.

[22] R. J. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[23] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.

[24] J. Crow and B. L. Di Vito. Formalizing space shuttle software requirements. In *Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice*, pages 40–48, January 1996.

[25] W. Damm, H. Hungar, P. Kelb, and R. Schlör. Statecharts: Using graphical specification languages and symbolic model checking in the verification of a production cell. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*, pages 131–149. Springer-Verlag, 1995.

[26] D. L. Dill. The Murφ verification system. In Alur and Henzinger [4], pages 390–393.

[27] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of FMSP'98: The 2nd Workshop on Formal Methods in Software Practice*, pages 7–15, Clearwater Beach, Florida, USA, March 1998.

[28] Federal Aviation Administration, US Department of Transportation. *Introduction to TCAS II*, March 1990.

[29] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[30] O. Grumberg, editor. *Computer Aided Verification, 9th International Conference, CAV'97 Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

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

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

[33] M. P. E. Heimdahl and B. J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *Proceedings of the IEEE High Assurance Systems Engineering Workshop*, Niagara Falls, Canada, October 1996.

[34] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[35] J. Helbig and P. Kelb. An OBDD-representation of statecharts. In *Proceedings: The European Design and Test Conference. EDAC, The European Conference on Design Automation. ETC, European Test Conference. EUROASIC, The European Event in ASIC Design*, pages 142–149, Paris, France, February/March 1994. IEEE.

[36] K. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1):2–12, January 1980.

[37] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings: 36th Annual Symposium on Foundations of Computer Science*, pages 453–462, Milwaukee, Wisconsin, USA, October 1995. IEEE.

[38] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. In Grumberg [30], pages 460–463.

[39] D. Jackson. Abstract model checking of infinite specifications. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods, 2nd International Symposium of Formal Methods Europe, Proceedings*, volume 873 of *Lecture Notes in Computer Science*, pages 519–531, Barcelona, Spain, October 1994. Springer-Verlag.

[40] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–95, July 1996.

[41] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[42] O. Kupferman and M. Y. Vardi. Module checking. In Alur and Henzinger [4], pages 75–86.

[43] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley Longman, 1995.

[44] 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.

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

[46] K. L. McMillan. Fitting formal methods into the design cycle. In *31st ACM/IEEE Design Automation Conference, Proceedings 1994*, pages 314–319, San Diego, USA, June 1994.

[47] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.

[48] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [4], pages 411–414.

[49] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS'91*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264, Sendai, Japan, September 1991. Springer-Verlag.

[50] 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*, volume 1051 of *Lecture Notes in Computer Science*, pages 425–444, Oxford, UK, March 1996. Springer-Verlag.

[51] J. D. Reese and N. G. Leveson. Software deviation analysis: A "safeware" technique. In *Proceedings of the 1997 International Conference on Software Engineering: ICSE 97*, pages 250–260, Boston, USA, May 1997.

[52] E. M. Sentovich. A brief study of BDD package performance. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design: 1st International Conference, FMCAD'96 Proceedings*, pages 389–403, Palo Alto, California, USA, November 1996. Springer-Verlag.

[53] 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.

[54] J. S. Thathachar. On the limitations of ordered representations of functions. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV'98 Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 232–243, Vancouver, Canada, June/July 1998. Springer-Verlag.

[55] J. M. Wing and M. Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28(2/3):273–299, April 1997.

[56] J. Yang, A. K. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. *ACM Transactions on Programming Languages and Systems*, 19(2):386–412, March 1997.