Model Checking Large Software Specifications

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

Technical Report 96-04-02

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
May 3, 1996

Abstract

In this paper we present our results and experiences of using symbolic model checking to study the specification of an aircraft collision avoidance system. Symbolic model checking has been highly successful when applied to hardware systems. We are interested in the question of whether or not model checking techniques can be applied to large software specifications.

To investigate this, we translated a portion of the finite-state specification of TCAS II (Traffic Alert and Collision Avoidance System) into a form accepted by a model checker (SMV). We successfully used the model checker to investigate a number of dynamic properties of the system.

We report on our experiences, describing our approach to translating from RSML to SMV and our methods for achieving acceptable performance in model checking, and giving a summary of the properties that we were able to check. We consider the paper as a data point that provides reason for optimism about the potential for successful application of model checking to software systems. In addition, our experiences provide a basis for characterizing features that would be especially suitable for model checkers built specifically for analyzing software systems.

The intent of this paper is to evaluate symbolic model checking of state-machine based specifications, not to evaluate the TCAS II specification. We used a preliminary version of the specification, the version 6.00, dated March, 1993, in our study. We did not have access to later versions, so we do not know if the properties identified here are present in later versions.

*Email addresses: {anderson, beame, burns, wchan, fm, notkin, jdreese}@cs.washington.edu. Contact authors: Richard Anderson and David Notkin.

1 Introduction

Model checking is a technique for analyzing finite state spaces. Most model checkers take as input a finite state machine and a temporal predicate about that state space and produce as output either a counterexample or else an assertion that the predicate is true. Model checking is now widely and successfully used for verifying properties of hardware systems, even for many with exceedingly large state spaces.

The question of whether model checking can be applied as profitably to software systems as to hardware systems remains open. One key concern is that model checking is limited to handling finite state machines even though software systems are generally specified as infinite state machines. Jackson [14] and Wing and Vaziri-Farahani [20] have addressed aspects of this concern, showing some techniques for approximating infinite state machines with finite state machines that can then be used for model checking.

A second key concern is that hardware systems tend to possess certain properties, such as regularity, that allow model checking to succeed, while software systems may not exhibit similar properties. This concern has not been as aggressively addressed in the literature. The examples used by Jackson, by Wing and Vaziri-Farahani, and by Atlee and Gannon [3], for instance, are useful for showing the basic promise of applying model checking to software systems, but still consider relatively small systems. When the systems and their associated state spaces grow in size, the question remains as to whether they will exhibit structures and properties conducive to model checking. In this paper, we address this question in greater detail by discussing our effort in applying model checking to a large and realistic finite state specification.

Specifically, we manually translated a significant portion of a preliminary version of the TCAS II (Traffic Alert and Collision Avoidance System) System Requirements Specification from the Requirements State Machine Language (RSML) [16] into a form suitable for input to the Symbolic Model Verifier (SMV) [17]. We then wrote formulae to check a number of properties of the specification. These included properties that had been identified previously by other techniques, as well as properties of the specification that were not known to any of the authors before model checking was applied. We were able to study properties relating to the consistency of transitions and the consistency of function definitions, as well as safety properties. These results are detailed in Section 5.

Recently, Sreemani and Atlee [19], in work independent of ours, analyzed the A-7E aircraft software requirement specification with SMV, and were also able to check several dynamic properties. While their motivations were similar, our studies differ in several ways because of differences in the specifications. The A-7E aircraft requirements were written in the Software Cost Reduction (SCR) requirements notation [1, 13], as opposed RSML. RSML contains features like hierarchical states and particular assumptions and constraints on timing that are not present in SCR. It was previously believed that a state hierarchy has to be flattened, or the analysis has to be limited to a subset of the RSML features [2]. In addition, the environment of the A-7E specification is abstracted as a set of predicates, whereas the inputs to our system are events and numerical values. Numerical calculation and comparison are abundant in the TCAS specification, and they could incur significant performance problems in the model checking process.

Our objective was to test the effectiveness of model checking technology on software systems, so our experiences in applying model checking are more important than the individual results. We convey some of the obstacles we faced and the techniques that we used to overcome these

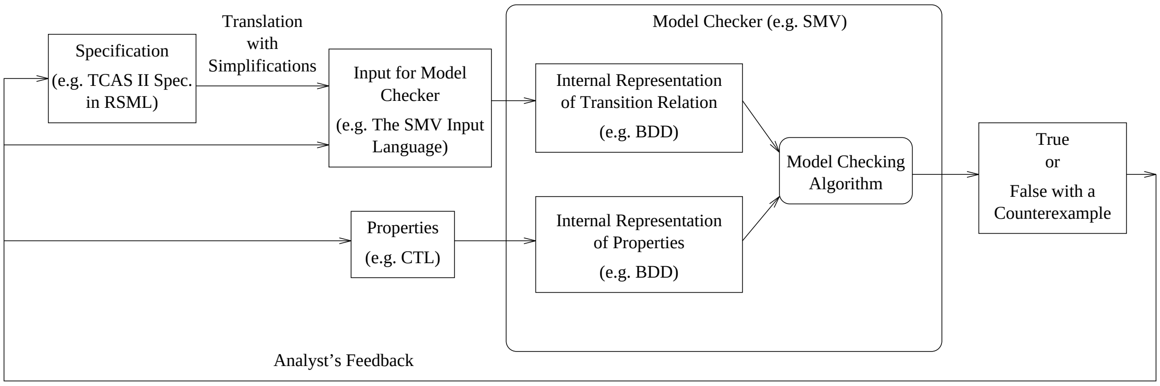

Figure 1: Model Checking

obstacles to allow us to check formulae against the specification. Other software systems that are often specified using finite state machines — for example, telephony and communication systems, network and distributed system protocols, and other reactive systems — might well yield to similar analyses. In this domain, the question of producing finite state specifications from infinite state specifications is largely moot, since finite state specifications are already ubiquitous.

Based on our experience, and as an additional step towards making model checking of software specifications more practical, we discuss some of the limitations of current model checking technology and suggest directions for developing model checkers better suited to software.

2 Model Checking

Many systems can be usefully modeled as finite state machines governed by transition relations. We can express properties about the state space as formulae in a temporal logic. The truth value of the formula could in principle be determined by exploring the reachability graph of the state space. We use the term *model checking* to mean exploring a finite state space to establish properties of the system. However, the state spaces arising from practical problems are often huge, so exhaustively exploring the state space is not generally feasible. What is done is to use an *implicit* representation of the state space, where the reachability relation is represented as a collection of boolean formulae. The truth values of temporal formulae are tested by a series of operations on the boolean formulae as opposed to an explicit search. This is referred to as *symbolic model checking*.

Figure 1 is a schematic of the model checking process with the specific representations that we used for the components shown in parentheses. The specification is translated to an input for the model checker, possibly with some simplifications of the model. The input and the formula that is being tested are then converted to the internal representation of the model checker. The representations are passed to the model checking algorithm. The result is either a claim that the formula is true or else a counterexample that shows that it is false. The result can be analyzed by the software engineer to refine the model of the specification, the temporal formulae, or even the specification itself. This iterative process will be further illustrated later.

Since we are interested in dynamic properties, the formulae are usually expressed in a temporal logic. A logic in widespread use is Computational Tree Logic (CTL)[9], a branching-time temporal logic. CTL is designed to conveniently express both safety and liveness properties. In addition to

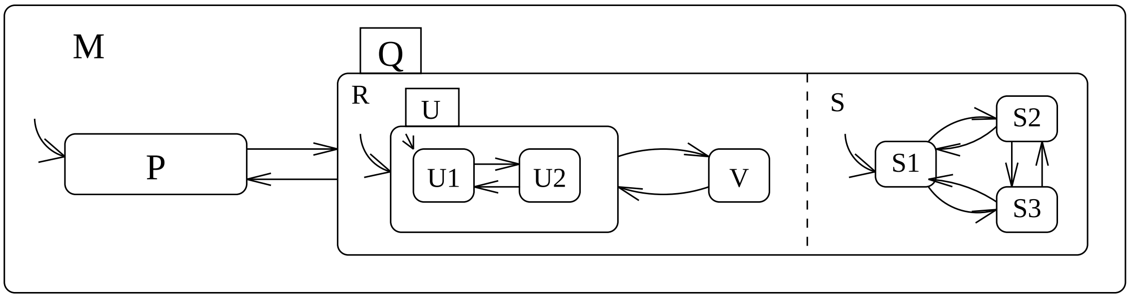

Figure 2: An Example of an RSML state machine.

standard logical symbols, there are next, until, future, and global operators. The examples in this paper use a global operator G, so that the formula `AG (x | (y & !z))` says that in all reachable states either x is true, or y is true and z is simultaneously false.

A data structure that has been developed to represent boolean functions is the *Ordered Binary Decision Diagram* (OBDD, or BDD for short) [6]. A BDD is a directed acyclic graph that encodes the function. (One way to view it is as a decision tree with isomorphic sub-trees identified.) The properties that make BDDs useful in model checking include that they give unique representation of functions, they can be combined efficiently and there are algorithms that can manipulate BBDs to test logical relations. Several hardware model checkers, such as SMV which we used in our study, have been constructed using BDDs as their internal representation. These are successfully used for checking large circuits in both commercial and academic settings. The key for these checkers to work efficiently is that the BDD representation remains small even when the state space being explored is very large.

A BDD is defined with respect to a fixed ordering of variables, and all the BDDs in use must use the same ordering of variables. Many operations have concise BDD representations under all variable orderings. Some operations such as addition and comparison of integers have concise representations only with a suitable variable ordering. However, some operations, such as multiplication, cannot be represented concisely, requiring exponential size under any variable ordering ([4, 18]).

3 Translating RSML Specifications into SMV programs

Before we could apply the BDD model checking algorithms to the TCAS specification, we had to first translate the specification from RSML into a form accepted by a BDD based model checker, such as SMV. We first briefly overview RSML and SMV, which lays the foundation for our description of the translation.

3.1 RSML

RSML is based on a Mealy-machine model of finite-state machines with outputs on the transitions between states. RSML includes several features found in other communicating state machine models (e.g., Statecharts [11]), such as hierarchical abstraction into superstates and parallel state machines (AND decomposition). It also has innovative features like transition buses and AND/OR tables.

Figure 2 is an example of an RSML state machine. It shows the state hierarchy and the transitions between the states. There are three kinds of states in RSML: OR states, in which

Transition(s): $\boxed{\text{S1}} \longrightarrow \boxed{\text{S2}}$

Location: M $\triangleright$ Q $\triangleright$ S

Trigger Event: x

Condition:

		OR	
A	R **in state** U	T	.
N	Alt > 1000 ft	T	.
D	t $\geq$ t(entered(Q)) +5 sec	.	T

Output Action: y

Figure 3: Transition from S1 to S2.

exactly one substate is active at any given time (e.g. M, whose substates are P and Q), AND states, in which all the substates are executed in parallel (e.g. Q, whose substates are R and S), and atomic states (e.g. P), which have no substates. A substate of an AND state or an OR state can be an AND state, an OR state or an atomic state. In the figure, arrows without origins specify start states. For example, when state Q is entered, the machine is in states U1 and S1.

A transition consists of a source state, a destination state, a trigger event, and possibly a guarding condition and/or an output action. A transition is taken when its trigger event occurs and its guarding condition (if present) is true, thus producing an output action. The action identifies an event that may trigger another transition in the system. The guarding conditions on a transition are expressed in a tabular representation of disjunctive normal form called AND/OR tables (see Figure 3.) The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. The table evaluates to true if one of its columns is true. A column evaluates to true if all of its entries are true. A dot denotes "don't care." When two or more transitions out of a state are triggered simultaneously leading to different next states, the state transition is nondeterministic. Nondeterminism is usually a design flaw in the specification [15].

Figure 3 shows a possible transition from S1 to S2. The transition is taken exactly when trigger event x is generated and the predicate specified by the AND/OR table is true. Event x may be triggered by some other transition in the system, or by the input interface as a result of receiving an external message from the environment. In the AND/OR table, t is a special variable in RSML that indicates the current time, while t(entered(Q)) is a function that returns the time when state Q was last entered. Therefore, the AND/OR table specifies the predicate that either (column 1) state R is in U and Alt is greater than 1000 ft or (column 2) the machine entered state Q at least 5 seconds ago. Alt can be an input variable, or an arbitrary function. If the transition is taken, event y will be generated, possibly triggering other transitions in the machine.

The cascading of events continues until no transitions are generated. At this point, the system reaches a *stable* state. A *step* is defined by the change in the system state from the point at which the initial event was received until the point when system reaches a stable state. Each interim state change in a step is called a *microstep*. A step (and thus a microstep) is assumed to happen instantaneously. Once a step is initiated, no external messages can arrive until the system reaches a stable state. This assumption is called the *synchrony hypothesis* [16].

3.2 SMV

SMV is a tool for checking finite state systems against specification written in the temporal logic CTL. SMV supports specification of synchronous Mealy machine systems as well as specification of asynchronous finite state systems. It also supports both deterministic and nondeterministic models, and provides for modular system descriptions. Since SMV is meant to describe finite state machines, it contains only boolean, scalar and fixed array data types. Below we summarize only the SMV features pertinent to our discussion.

An SMV program is divided into modules, each of which consists of variable declarations and assignments, and a list of CTL formulae. Transitions in SMV are defined by a collection of parallel assignments to variables. The assignments are simultaneously executed once at each clock tick. The statement **init(***var***)** sets the initial value of variable *var* and **next(***var***)** sets its value in the next state. The statement **next(***var***)** can contain a **case** expression, so that transitions can have guarding conditions.

A variable can be nondeterministic. There are two sources of nondeterminism in SMV. An expression can be a set, and it nondeterministically evaluates to a value from that set. In addition, when the initial or the next state value of a variable is not defined, SMV nondeterministically assigns it a value of its type.

An expression can also be assigned to a symbol instead of a variable. In this case, a variable is not introduced in the BDD representation of the system. Instead the symbol's definition is expanded when it is used. It is analogous to `#define` in C.

Properties to model check are expressed as CTL formulae. SMV uses symbolic model checking algorithms based on BDD to determine whether the CTL formulae are satisfied by the SMV program.

3.3 Translating RSML to SMV

In this section we present an overview of the rules we derived in translating RSML specifications into SMV programs.

The Synchrony Hypothesis In our example in Figure 3, suppose that x is an event that is generated as a result of receiving an external message. We declare a boolean variable `x` for event x. We call such a variable an *environment variable*. We allow SMV to nondeterministically assign values to the environment variables to model the unpredictable inputs. However, to maintain the synchrony hypothesis, we have to restrict them to change only when the system is stable. So, we first define a symbol `Stable`, which is a conjunct of the negation of all the variables that represent events. That is, we define

```
Stable := !x & !y & !z;
```

assuming `x`, `y` and `z` are the only events in the system. For event x we have an assignment:

```
next(x) := case
             Stable: {0,1};   -- line 1
             1: 0;            -- line 2
           esac;
```

(Comments in SMV start with "--".) The statement specifies that (line 1) if the system is stable in the current state, the value of `x` is 0 or 1 nondeterministically in the next state; (line2) otherwise, the value of `x` is 0 in the next state, meaning that event `x` has been received in the current microstep. (1 means true in SMV, so a guarding condition of 1 serves as the default case.)

This nondeterministic setting of the values of environment variables is a simplification of how the environment operates. Of course there are certain assumptions on changes in inputs that are necessary for the correct behavior of the system. If the assumptions are known, we can model them by specifying how the environment variables change values. However, allowing SMV to nondeterministically set the variables enables us to examine the effects of violating these assumptions on properties of the system.

Hierarchical States and Transitions To translate a state hierarchy into SMV, we create an SMV variable for each OR state in the hierarchy that either is a root state or is a child state of an AND state. The type of such a state variable is an enumerated set containing one value for each of its descendent states that is either an AND state or an atomic state and is not a descendant of some other state variable.

In our example, we declare:

```
M: {P, Q};
R: {U1, U2, V};
S: {S1, S2, S3};
```

The values of the three variables completely describe which state the machine is in. The change of these variable solely depends on state transitions of the machine.

A transition is taken if and only if (1) the machine is in the source state of the transition, (2) the trigger event occurs, and (3) the guarding condition specified by the AND/OR table is satisfied. We define a symbol for each transition. It is assigned a boolean expression, which is a logical conjunction of the above three conditions. For the transition in Figure 3, we define:

```
T_S1_S2 := (M = Q) & (S = S1)                              -- source state
           & x                                             -- trigger event
           & (((M = Q & (R = U1 | R = U2)) & Alt > 1000)   -- guarding condition (column 1)
              | Time_Since_Entered_Q >= 5);                -- guarding condition (column 2)
```

Notice that the first line of the expression is `(M = Q) & (S = S1)` instead of simply `(S = S1)`. The reason is that if the machine is not in state Q, then it is not in state S1 regardless of the value of the variable `S`. The third line of the expression shows how to translate the condition that the machine is in state U: the machine is in state Q, and it is either in state U1 or in state U2. The variable `Time_Since_Entered_Q` will be explained shortly. So, to model the state change, we have an assignment:

```
next(S) := case
             !(M = Q) & next(M) = Q : S1;    -- line 1
             T_S2_S1 | T_S3_S1 : S1;
             T_S1_S2 | T_S3_S2 : S2;
             T_S1_S3 | T_S2_S3 : S3;
             1 : S;
           esac;
```

where T_S2_S1, T_S3_S1, etc. would be defined similarly to T_S1_S2. Notice that line 1 in the case expression specifies that the start state of S is S1.

Output actions are modeled as a logical disjunction of the transitions that generate it. For example:

```
next(y) := T_S1_S2 | T_U1_U2;
```

assuming the transitions from S1 to S2 and from U1 to U2 are the only transitions that trigger event y.

This translation is not precise when two of the transitions, say T_S1_S2 and T_S1_S3, can be triggered simultaneously: the semantics of RSML are that the transition is nondeterministic in this case, whereas SMV always evaluates the conditions in a **case** expression in order. Nondeterminism is usually unintentional, and in Section 5.1 we will describe how to detect the inconsistency. If the nondeterminism is intentional, we can model it by declaring an unassigned (and thus nondeterministic) variable and including it in the case expression. For example, we can declare a boolean variable coin, and after line 1 in the **case** expression above we can add:

```
T_S1_S2 & T_S1_S3 & coin : S2;
T_S1_S2 & T_S1_S3 & !coin : S3;
```

These two conditions state that if the two transitions are triggered simultaneously, the machine will go to S2 if coin is 1, and go to S3 otherwise.

Timing constraints Recall that in Figure 3, there is a timing constraint t $\geq$ t(entered(Q)) + 5 sec, which is equivalent to t $-$ t(entered(Q)) $\geq$ 5 sec. So, to model this constraint, we need the difference between the current time and the time at which state Q was last entered. We create an variable Time_Since_Entered_Q to implement a timer:

```
next(Time_Since_Entered_Q) := case
                                 !(M = Q) & next(M) = Q : 0;
                                 Stable & Time_Since_Entered_Q < 5 : Time_Since_Entered_Q + 1;
                                 1: Time_Since_Entered_Q;
                              esac;
```

The assignment says that (line 1) if the machine enters state Q, reset the timer, (line 2) if the machine is stable and the timer is less than 5 seconds, advance the timer and (line 3) otherwise, the timer remains unchanged.

Notice that this implementation assumes that arrivals of inputs are separated by multiples of one second. This assumption happens to be also true in TCAS. If the time granularity is different, we can simply scale the constants accordingly.[1]

Miscellaneous Our example does not contain all RSML constructs, such as PREV(), constants, macros, functions, statechart arrays, and transition buses. Roughly, PREV(e) returns the previous value of expression e. Modeling PREV() requires introducing an auxiliary variable to "remember" the variable's previous state. Constants can be trivially implemented with defined symbols in SMV. Macros and functions without arguments can be modeled similarly. Macros and functions with arguments are somewhat trickier; they can be implemented as SMV modules that are instantiated at each call site. Statechart arrays can be implemented as an array of modules. The translation of transition buses is no different from that of ordinary transitions.

[1]We assume that time is discrete.

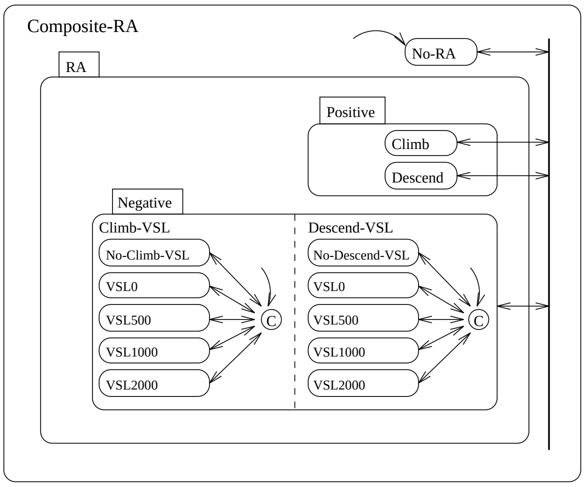

Figure 4: Composite_RA in Own_Aircraft of TCAS

4 Obstacles to Model Checking TCAS with SMV

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

4.1 TCAS

The first obstacle we had to overcome was the sheer size of the preliminary TCAS II System Requirements Specification, a 400 page document. As a first attempt we decided to try model check a portion of it, i.e. a state machine called Own_Aircraft, which occupies about 30% of the specification. Own_Aircraft has close interactions with another part of TCAS called Other_Aircraft. Own_Aircraft models the state of the TCAS equipped aircraft under consideration (the own aircraft). Other_Aircraft tracks the state of some other aircraft in the vicinity of the own aircraft, and gives a resolution advisory (RA) to Own_Aircraft. Up to 30 other aircraft can be tracked. Considering the advisories given by the 30 instances of Other_Aircraft, Own_Aircraft derives a composite resolution advisory and generates visual and audio outputs to the pilot. Examples of advisories include Climb, Descend, Increase-Climb ("increase the current climb rate"), Increase-Descend, Climb-VSL0 ("do not climb"), Climb-VSL500 ("do not climb more than 500 ft/min"), etc. Figure 4 shows the state Composite_RA, one of the twelve parallel substates of Own_Aircraft.

We treated Other_Aircraft as part of the environment of Own_Aircraft. That is, we created environment variables for any states of Other_Aircraft that are referenced within Own_Aircraft. Like other environment variables they were nondeterministic, except that we restricted when these variables could change to ensure correct synchronization behavior. We focused on resolution maneuvers with one intruder and thus modeled only one instance of Other_Aircraft.

4.2 BDDs

We knew a priori that there is no efficient BDD representation for multiplication and division so we realized that 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 variables that we are already modeling nondeterministically so we made the conservative simplification to treat these calculated values as nondeterministic themselves. This simplification did not cause problems for the properties that we checked and reported below.

4.3 SMV

The performance of BDD based algorithms is directly related to the size of the BDDs that are generated. 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 formulae with the large BDDs were generally unsuccessful or too slow (our initial success in identifying nondeterminism was an overnight run, although we can now find the nondeterminism in a few minutes).

The size of the BDDs can be reduced by variable reordering and conjunctive partitioning [7]. These techniques dramatically improved the performance of checking some formulae; 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. Initially, we had to limit the precision of all the variables to at most 4-bit numbers and redefine the constants accordingly in order for the model checker to build the BDDs in a reasonable amount of time. Increasing the ranges by one bit sometimes exploded the checking time from ten minutes to more than ten hours.

The problem had to do with the issue of variable ordering for BDDs representing integer addition and comparison mentioned earlier. In particular, a BDD for the sum of the integers $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_n$ has size $O(n)$ if the variables are in the order $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ but requires exponential size if the variables are in the order $x_1, x_2, \ldots x_n, y_1, \ldots, y_n$. SMV does not interleave the bits among the variables it is representing when constructing the BDDs. Therefore, although comparison and addition have concise BDD representations, SMV produces exponential size BDDs for them.

We considered two ways of attacking this problem, namely changing the internals of SMV to interleave the bits, or doing addition and comparison at the source code level. The latter method seemed a simpler approach and we were able to use it with great success, although in principle the former may be a better long term solution. We implemented our own adder and comparator in our SMV program and manipulated all the variables and constants at the bit level.[2] Changing the variables in our TCAS translation from 4-bit numbers to 15-bit numbers blows up the size of the state space roughly from 10^{40} to 10^{64}. However with our source code level implementation of addition and comparison, this increase in precision increased the run time and the number of BDD nodes used by less than a factor of three. Variables with 15 bits are enough for representing all the integers in TCAS.

Yet another performance problem was that generating a counterexample often took hours even though the formula was determined false within minutes. To tackle this problem, we optimized the counterexample searching routine in SMV, so that once a formula is evaluated false, a counterexample that falsifies the formula can now be found almost instantly.

[2]We did this in a disciplined way by writing some awk scripts to preprocess the SMV program.

5 Results of Model Checking TCAS

Once we overcame these obstacles, we were positioned to do some analysis of the preliminary TCAS specification using SMV. We report on some of the results below. For each of the properties below, the model checker generated the result (with a counterexample if the formula was false) within several minutes on a Sun SPARCstation 10. The process consumed less than 40MB out of the 256MB physical memory. The size of the *reachable* state space was at least 10^{56} as reported by SMV.

5.1 Transition Consistency

There are known nondeterministic transitions in earlier versions of the TCAS specification. So, our first attempt at validation was to find such transitions in one of these versions with the model checker. (For the other properties that we checked, we worked with a later draft TCAS specification, in which there is no unintentional nondeterminism.) These nondeterministic transitions had previously been identified by Heimdahl and Leveson[12] using static analysis. We were interested in checking these properties to verify that model checking could match previous results. Static techniques are conservative in that the problematic states may not be reachable (although in this particular case it is obvious that they are). Model checkers also generate counterexamples, giving proofs that the states are indeed reachable from some initial state.

In our example in Figure 2, possible nondeterministic transitions are in state S. For example, the transitions from S1 to S2 and from S1 to S3 would be triggered at the same time if their trigger events were the same and their guarding conditions were simultaneously satisfied. We can check this with the model checker by the following CTL formula:

```
AG !(T_S1_S2 & T_S1_S3)
```

Recall that `T_S1_S2` is true when the transition from S1 to S2 is taken; similarly for `T_S1_S3`. So the CTL formula specifies that the two transitions are never taken simultaneously. Applying this technique to all the states, the model checker was able to find the known nondeterministic transitions in that particular version of the TCAS specification.

5.2 Function Consistency

Displayed_Model_Goal, shown in Figure 5, is a function in the TCAS specification whose value is displayed to the pilot. It represents the optimal altitude rate at which the pilot should aim. 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 is similar to checking for nondeterminism. We define a boolean symbol `Case-1` for the first Case, and `Case-2` for the second case, and so on, and check an CTL formula of the form:

```
AG !((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

$$
\text{Displayed\_Model\_Goal} = \begin{cases}
0 & \textbf{if } \text{Composite\_RA } \textbf{not in state } \text{Positive} \quad \text{/* case 1 */} \\[1ex]
\begin{array}{l}\textbf{Max}(\text{Own\_Track\_Alt\_Rate}, \\ \text{PREV(Displayed\_Model\_Goal)}, \\ \text{1500 ft/min})\end{array} & \begin{array}{l}\textbf{if } (\text{New\_Climb } \textbf{or } \text{New\_Threat}) \textbf{ and} \quad \text{/* case 2 */} \\ \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}\end{array} \\[1ex]
\begin{array}{l}\textbf{Min}(\text{Own\_Track\_Alt\_Rate}, \\ \text{PREV(Displayed\_Model\_Goal)}, \\ -\text{1500 ft/min})\end{array} & \begin{array}{l}\textbf{if } (\text{New\_Descend } \textbf{or } \text{New\_Threat}) \textbf{ and} \quad \text{/* case 3 */} \\ \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}\end{array} \\[1ex]
\text{2500 ft/min} & \textbf{if } \text{New\_Increase\_Climb} \quad \text{/* case 4 */} \\[1ex]
-\text{2500 ft/min} & \textbf{if } \text{New\_Increase\_Descend} \quad \text{/* case 5 */} \\[1ex]
\begin{array}{l}\textbf{Max}(\text{Own\_Track\_Alt\_Rate}, \\ \text{1500 ft/min})\end{array} & \begin{array}{l}\textbf{if } \text{Increase\_Climb\_Cancelled } \textbf{and} \quad \text{/* case 6 */} \\ \textbf{not } \text{New\_Increase\_Climb } \textbf{and} \\ \text{Composite\_RA } \textbf{in state } \text{Positive}\end{array} \\[1ex]
\begin{array}{l}\textbf{Min}(\text{Own\_Track\_Alt\_Rate}, \\ -\text{1500 ft/min})\end{array} & \begin{array}{l}\textbf{if } \text{Increase\_Descend\_Cancelled } \textbf{and} \quad \text{/* case 7 */} \\ \textbf{not } \text{New\_Increase\_Descend } \textbf{and} \\ \text{Composite\_RA } \textbf{in state } \text{Positive}\end{array} \\[1ex]
\text{PREV(Displayed\_Model\_Goal)} & \text{Otherwise} \quad \text{/* case 8 */}
\end{cases}
$$

Figure 5: Definition of Displayed_Model_Goal in the TCAS specification. Most of the identifiers are RSML macros or abbreviations, the definitions of which are omitted here due to limited space. (Their truth values depend on Composite_RA and Other_Aircraft.)

counterexample, Other_Aircraft reverses from an Increase-Climb advisory to an Increase-Descend advisory 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.

5.3 Safety Properties

The state of Composite_RA in Figure 4 is also shown to the pilot. Therefore it is critical that Composite_RA and Displayed_Model_Goal are consistent with each other. 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 the preliminary specification. In fact, it showed that when Composite_RA is Climb, Displayed_Model_Goal could be negative. The CTL formula we checked is similar to the following:[3]

```
AG !(Composite_RA = Climb & Displayed_Model_Goal < 1500)
```

The counterexample given by the model checker is a three step scenario:

1. At time t_0, there is an intruder aircraft and Other_Aircraft gives a Descend advisory. As a result, Composite_RA is in state Descend and by case 3 of the definition of Displayed_Model_Goal, it is ≤ -1500 ft/min.

[3]The exact formula differs due to some implementation details. The same is true for the other formula below.

2. At time $t_1 > t_0$, Other_Aircraft realizes that an increase in descend rate is necessary and issues an Increase-Descend advisory, 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 advisory to Climb, making Composite_RA enter state Climb. At that point, case 7 applies and Displayed_Model_Goal becomes ≤ -1500 ft/min, resulting in contradictory outputs.

Another property that was shown to be not satisfied is that, when Other_Aircraft changes to an Increase-Climb advisory, Displayed_Model_Goal should not decrease. In CTL this is:

```
AG !(New_Increase_Climb & Displayed_Model_Goal < Prev_Displayed_Model_Goal)
```

where `New_Increase_Climb` evaluates to true when Other_Aircraft gives a new Increase-Climb advisory, and `Prev_Displayed_Model_Goal` is the previous value of Displayed_Model_Goal. However, the counterexample it gave required an Increase-Climb advisory to be generated when the value of Own_Track_Alt_Rate is greater than 2500 ft/min. This is prevented by the logic in the specification of Other_Aircraft, which we had modeled as a part of the nondeterministic environment. After refining our model of Other_Aircraft to avoid generating an Increase-Climb advisory in such a situation, the formula was shown to be true by the model checker.

Yet another safety property is that when the aircraft is close to the ground, Displayed_Model_Goal should never be less than some threshold. Specifically, whenever the aircraft is below 1450 feet above ground level, Displayed_Model_Goal ≥ -2500 ft/min. The model checker found that this property was not satisfied. The counterexample it gave revealed a typographical error in the preliminary specification ($>$ instead of $\leq$).[4]

5.4 References to Uninitialized Values

It is possible for an AND/OR table or function to refer to the previous value of some variable (e.g., an input variable, state, or function reference) even though the variable was not yet defined in the previous step. In such a case the value of PREV() is undefined. The model checker handles such undefined references in the same way that it handles environment variables. That is, it nondeterministically assigns values in an attempt to find a counterexample to the formula. So while analyzing for the properties mentioned above, the model checker also discovered situations where a variable is referenced before it is defined, e.g., referring to PREV() in the first step.

5.5 Discussion

Our translation rules are precise in that the set of transition sequences allowed by the RSML state machine is identical to that of the state machine specified by the translated SMV program. Therefore if the entire specification was translated, the model checking would be both sound and complete. However, because we abstracted Other_Aircraft, some counterexamples found by the model checker were not possible for the full specification. We had to incrementally constrain the model of Other_Aircraft to eliminate the spurious transitions.

[4] The authors had discovered the typographical error by observation during the translation process.

It may seem that the repeated process of getting an incorrect counterexample and fixing it is an undesirable artifact of the incomplete translation of the specification. There are several reasons why leaving part of the model nondeterministic is in fact a useful technique:

- A specification may be so complex that model checking it in its entirety is infeasible. This approach, then, allows model checking to be beneficially applied to parts of the specifications without fully considering all the remaining components.
- 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.
- Development and analysis of the specification can be interleaved so that potential problems can be found or avoided earlier. For example, when developing the TCAS specification, an engineer could first have specified Own_Aircraft and have left Other_Aircraft nondeterministic. An analyst could then have model checked Own_Aircraft and discovered the assumptions on the behavior of Other_Aircraft that are necessary for Own_Aircraft's correct operations. This information could then have been used to develop Other_Aircraft, which could be model checked later to see whether the assumptions hold.

This iterative approach appears to have benefits for analysis and shows potential for iterative development of specifications, as well.

6 Related Work

There are a number of other widely researched approaches to handling the state-space explosion problem. Corbett recently classified these techniques into several categories [10]: in addition to symbolic model checking, Corbett considered several variants on standard reachability analysis techniques, several compositional techniques, some approaches that exploit abstraction, some that use data flow analysis, and some that use integer programming. We have not yet done a comparison of our use of model checking to any of these techniques.

7 Conclusions and Future Work

We have shown that it is feasible to translate part of a large finite state specification into a form suitable for a model checker, and have been able to check several non-trivial properties. Our approach to analyzing the specification incrementally, by modeling some components nondeterministically and then refining them, proved to be quite powerful. We believe that these are important steps towards realizing symbolic model checking as an effective tool in the process of analyzing software specifications.

Additional research is needed to assess the strengths and weaknesses of the techniques that we used to overcome a number of obstacles in model checking parts of the TCAS specification:

- Multiplication and division cannot be represented efficiently by BDD's ([4, 18]). Bryant and Chen [5] introduced a different data structure, the BMD (Binary Moment Diagram) which can be used to represent multiplication concisely. With a variant of this data structure, the

*BMD, they were able to verify division circuits. A hybrid approach where BMD's are used to represent arithmetic variables and BDD's are used to represent control variables, as suggested by Clarke and Zhao [8], may be attractive. Building model checkers that can handle arbitrarily complicated numeric calculations is almost certainly intractable. However, rudimentary arithmetic, coupled with an understanding of the appropriate notions of approximation might be sufficient to handle many applications.

- Automating the translation from RSML to input to SMV (or another model checker) appears to be a straightforward matter of programming. It might be reasonable to develop a model checker that directly accepts languages such as RSML or Startcharts, eliminating the need for any source-level translation at all.

- It might be possible to exploit the general structure of the derived transition relation to improve performance. (Although we only showed how to translate the TCAS specification, we believe that this is a generalizable approach.) Our SMV description of an RSML specification had variables to represent the state space, time, environment, and internal events. Although we treated these uniformly in our translation to SMV, they were used in different ways. It is possible that a model checker that incorporated some of the semantics of time into the internal algorithms could outperform a checker that handled time with ordinary numeric variables. Similarly, testing whether the system had achieved stability could be subsumed by reachability algorithms. In other words, by exploiting common properties of software specifications that represent process control systems like TCAS, one might be able to build model checkers that perform better and are easier to use.

- We have not yet done a comparison of our use of model checking to a number of other approaches to handling the state-space explosion problem. Corbett recently classified these techniques into several categories [10]: in addition to symbolic model checking, Corbett discusses several variants on standard reachability analysis techniques, several compositional techniques, some approaches that exploit abstraction, some that use data flow analysis, and some that use integer programming. It would be interesting to compare these alternative approaches.

We believe that this investigation contributes to an increase in optimism that symbolic model checking can be successful in the analysis of software specifications.

Acknowledgments

We wish to thank the other members of the Winter 1996 CSE 590MC seminar at the University of Washington.

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 Lab., March 1988.

[2] J. M. Atlee and A. M. Buckley. A Logic-Model Semantics for SCR Software Requirements. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[3] J.M. Atlee and J. Gannon. State-based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, SE-19(1):24–40, January 1993.

[4] 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.

[5] R. E. Bryant and Chen Y.-A. Verification of arithmetic circuits with Binary Moment Diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 535–541, June 1995.

[6] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, August 1986.

[7] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[8] E. Clarke and X. Zhao. Work level symbolic model checking: A new approach for verifying arithmetic circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, May 1995.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-state Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–63, April 1986.

[10] J.C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering*, SE-22(3), March 1996.

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

[12] M.P.E. Heimdahl and N.G. Leveson. Completeness and Consistency Analysis of State-Based Requirements. In *Proceedings of the 17th International Conference on Software Engineering*, pages 3–14, April 1995.

[13] K. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[14] D. Jackson. Abstract Model Checking of Infinite Specifications. In *Proceedings of FME '94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe*, pages 519–31. Springer-Verlag, October 1994.

[15] 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.

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

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

[18] S. Ponzio. A lower bound for integer multiplication with read-once branching programs. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 130–139, May 1995.

[19] T. Sreemani and J. Atlee. Feasibility of model checking software requirements: A case study. Technical Report CS96-05, Department of Computer Science, University of Waterloo, January 1996.

[20] J.M. Wing and M. Vaziri-Farahani. Model Checking Software Systems: A Case Study. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, October 1995.