Model Checking Large Software Specifications

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

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

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 requirements 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 the specification to the SMV language 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.

1 Introduction

Model checking, a technique for analyzing finite state spaces, has been applied very successfully to a wide range of hardware systems. It has been surmised that there are two serious impediments that make it difficult to effectively apply modeling checking to software systems. The first possible impediment is that the technique is limited to handling finite state machines, while software systems are generally specified as infinite state machines. Jackson [15] and Wing and Vaziri-Farahani [22] 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. The second possible impediment—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—is the one we address in this paper. Specifically, we provide a data point by reporting on a positive experience in model checking a large software system requirements specification. With progress being made on these two fronts, it appears that applying model checking to software faces a brighter future than previously conjectured.

In our particular experiment, we translated (Section 3) a significant portion of a preliminary version of the TCAS II (Traffic Alert and Collision Avoidance System) System Requirements Specification [11] from the Requirements State Machine Language (RSML) [17] into a form suitable for input to the Symbolic Model Verifier (SMV) [18]. TCAS II is an aircraft collision avoidance system required on commercial aircraft with more than 30 seats, and was considered "the most complex system to be incorporated into the avionics of commercial aircraft" [17, p. 685]. We were able to generate an internal representation of the transition relation of the system of an acceptable size so that we could test a number of properties of the specification (Section 4). These include a number of general robustness properties as well as some safety properties specific to the domain (Section 5).

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 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. 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 (Section 7).

*Email addresses: {anderson, beame, burns, wchan, fm, notkin, jdreese}@cs.washington.edu.

To appear in the Fourth Symposium on the Foundations of Software Engineering, October, 1996, San Francisco, California, USA.

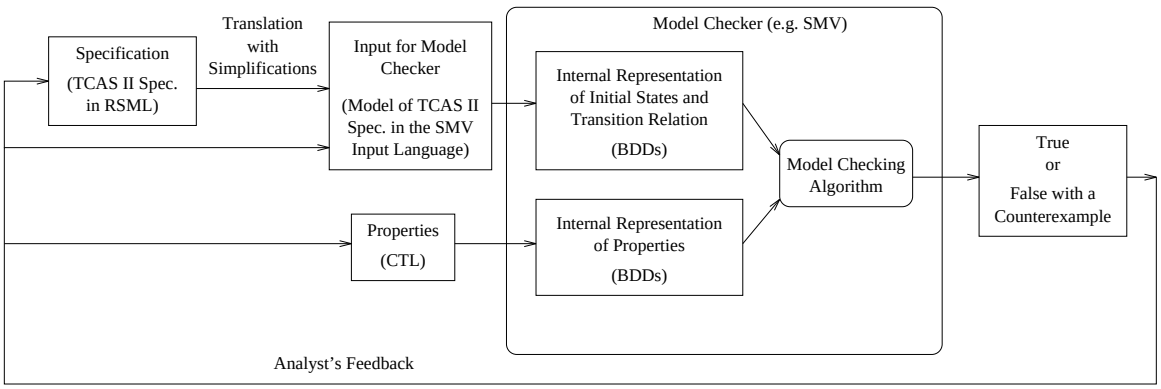

Figure 1: Model Checking a Specification

2 Model Checking

Model checking is the process of exploring a finite state space to determine whether or not a property holds. Figure 1 is a schematic of the process of model checking a specification, 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. The input and the property 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 property is true or else a counterexample (i.e. a sequence of state transitions starting from some initial state) showing that the property is false. The result can be analyzed by the software engineer to refine the model of the specification, the property tested, or even the specification itself. This iterative process is inherent in our work.

The major problem of model checking is that the state spaces arising from practical problems are often huge, generally making exhaustive exploration infeasible. An important advance in model checking was the introduction of symbolic representations of state spaces, which allowed direct exploration of the state space to be replaced by the manipulation of data structures representing the transition relation of the state space.

The transition relation can be represented as a boolean function. A data structure that has been developed to represent boolean functions is the *Ordered Binary Decision Diagram* (OBDD, or BDD for short) [5]. A BDD is a directed acyclic graph that encodes the function based on a fixed ordering of the variables. (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 a 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. This representation is frequently small although sometimes its size depends critically on the ordering of the variables.

Properties to be checked are usually expressed in a temporal logic, such as Computation Tree Logic (CTL) [8], which is used by SMV. CTL is a branching time temporal logic, extending propositional logic with temporal operators that express how propositions change their truth values over time. In this paper we will only use two temporal operators, namely AG and AF. Each CTL formula is evaluated with respect to some particular state. The formula AG p holds in state s if p holds in *all* states along all computation paths starting from s, and we call such a property an *invariant*. The formula AF p holds if p holds in *some* state along all computation paths starting from s. Therefore the formula AG ($p \rightarrow$ AF q) is true in state s if along all computation paths starting from s, whenever p is true, q will be true in some successor state along the path. CTL formulae are implicitly evaluated by SMV with respect to all the initial states.

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, laying the foundation for our description of the translation.

3.1 RSML

RSML is a communicating state machine model similar to Statecharts [12], including features such as parallel state machines (AND decomposition) and hierarchical abstraction into superstates. For the purposes of this paper, its most important semantic differences with Statecharts are a more restrictive interleaving (step) semantics and the separation of each trigger into a single positive triggering event and guarding conditions.

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 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

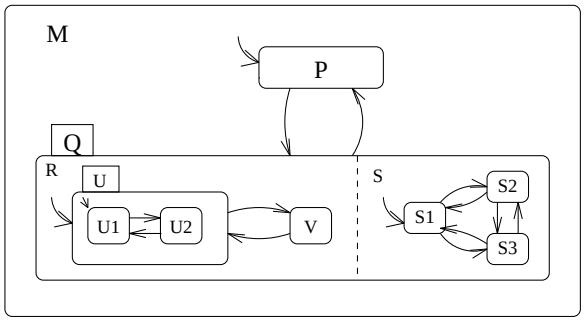

Figure 2: An Example of an RSML state machine.

Transition(s): S1 ⟶ S2

Location: M ▷ Q ▷ S

Trigger Event: x

Condition:

		OR	
AND	R **in state** U	T	.
	Alt > 1000 ft	T	.
	t ≥ t(entered(Q)) +5 sec	.	T

Output Action: y

Figure 3: Transition from S1 to S2.

can be an AND state, an OR state or an atomic state. In the figure, arrows without origins specify start states. For example, when the machine enters state Q, it 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 output 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 or output actions, the state transition is nondeterministic.

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 a 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 becomes *stable*. 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 becomes stable. Each interim state change in a step is called a *microstep*. A maximal set of mutually consistent transitions enabled at the start of each microstep fires simultaneously within that 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 becomes stable. This assumption is called the *synchrony hypothesis* [17].

3.2 SMV

SMV is a BDD-based tool for symbolic model checking of finite state systems against specifications written in the temporal logic CTL (see Section 2). It supports both deterministic and nondeterministic models, and provides for modular system descriptions. SMV contains 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 specifies a finite state machine. A module contains variable declarations to determine its state space and descriptions of the initial state and transition relation of the machine, as well as a list of CTL formulae to be checked. Variable declarations are preceded by the keyword `VAR`. The preferred method of describing the initial state is by a collection of parallel assignments to various **init**(*var*) where *var* is a variable. The expression **next**(*var*) is used to refer to the variable *var* in the next state. The preferred method of describing the transition relation is by a collection of parallel assignments to these **next** versions of the variables. The **init** assignments are made simultaneously at the start and the **next** assignments are simultaneously executed once per step. The values for these assignments can be based on a wide assortment of expressions. Assignments are preceded by the keyword `ASSIGN`.

SMV has a macro-like facility for defining a symbol to represent an expression. In this case, a variable is not introduced in the BDD representation of the system. In addition, SMV also extends the semantics of the **next** operator to apply to any expression *expr* that does not contain **next**. That is, **next**(*expr*) gives the value of expression *expr* in the next state. This is equivalent to replacing each variable *var* in *expr* by **next**(*var*). Symbols are defined after the keyword `DEFINE`.

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. 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.

SMV also has a somewhat more general but less robust way to specify the initial state and transition relation using INIT and TRANS constructs. These can be arbitrary propositional formulae involving the values of the variables, symbols, and their **init** and **next** versions. Although we did not use this feature in our translation of TCAS, it is useful for translating some RSML specifications, as we describe below.

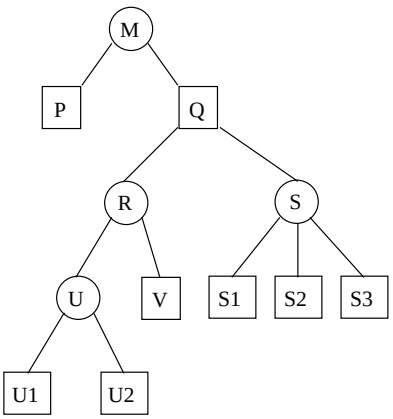

Figure 4: The state hierarchy drawn as a tree. The square nodes represent AND states and atomic states, and the round nodes OR states.

In SMV, 1 means true and 0 means false. The "and", "or" and "not" operators in SMV are `&`, `|` and `!` respectively.

3.3 Translating RSML to SMV

In this section we present an overview of the general methodology we derived for translating RSML specifications into SMV programs.

Hierarchical States One of the keys to successful use of symbolic model checking is to represent objects efficiently. In translating an ordinary finite state machine into SMV it is most efficient to represent the current state of the machine by a variable whose type is an enumerated set consisting of the possible machine states. This has the advantage of permitting the underlying BDD representation produced to be a binary encoding of the state space.

We extend this idea to a state hierarchy with parallel states in a natural way that preserves the alternation of the hierarchy but flattens nested OR states and nested AND states. More precisely, let $\mathcal{V}$ be the set consisting of the root state, if it is an OR state, together with all OR states in the hierarchy that are children of AND (parallel) states. For each state A, let $v(A)$ be its closest ancestor in $\mathcal{V}$ (if $A \in \mathcal{V}$ then $v(A) = A$.) Let $\mathcal{A}$ be the set consisting all atomic states together with all AND states in the hierarchy that are children of OR states.

We create one SMV variable for each element $\texttt{A} \in \mathcal{V}$. Its type is an enumerated set consisting of all elements $B \in \mathcal{A}$ such that $v(B) = A$. Continuing our example from section 3.1 and following Figure 4, we declare:

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

The values of these variables completely determine which states of the machine are current (because of the parallelism there may more than one current state.) For each state B we express whether or not state B is current by defining SMV symbols according to the following rules:

`inB := 1;` if B is the root state,

`inB := inA;` if B is a child of AND state A,

`inB := inA & (A = B);` if $B \in \mathcal{A}$ and $v(B) = A \in \mathcal{V}$,

`inB := inB1 | inB2 | ... | inBk;` if B is an OR state with children $B1, B2, \cdots, Bk$ and $B \notin \mathcal{V}$.

So for our state hierarchy, we have:

```
DEFINE
  inM := 1;
  inP := inM & (M = P);    inQ := inM & (M = Q);
  inR := inQ;              inS := inQ;
  inU := inU1 | inU2;
  inU1 := inR & (R = U1); inU2 := inR & (R = U2);
  inV := inR & (R = V);
  inS1 := inS & (S = S1); inS2 := inS & (S = S2);
  inS3 := inS & (S = S3);
```

Events, Input Variables, and the Synchrony Hypothesis Each RSML event x is represented by a boolean variable `x`. RSML input variables are translated directly as SMV variables. If the RSML input variable has an enumerated type or is an integer with a specified range, the translation is straightforward. If the RSML input variable is an integer and its range is not explicit, we set the range of the SMV variable to be sufficiently large to encompass the constants with which it and its functions are compared in the specification.

To model an unpredictable environment we allow SMV to nondeterministically assign values to the input variables. Of course there may be 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 input 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.

We simulate each microstep of RSML by a step in SMV. Therefore, to maintain the synchrony hypothesis of RSML we have to restrict the environment to change only when the system is stable. So, we define a symbol `Stable`, which is a conjunction of the negation of all the variables that represent events, and use it to guard all changes in input variables.

For example, assuming `x`, `y` and `z` are the only events in the system we define:

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

and, assuming that event `x` is generated by the environment, we assign:

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

In a case expression, the expression before a colon, e.g. `Stable`, serves as a guarding condition. If the guard evaluates to 1 (true), the case expression evaluates to the value of the expression after the colon, e.g. `{0,1}` (which in turn evaluates to 0 or 1 nondeterministically). The guards are considered in order. So the assignment specifies that `x` may be generated (set to 1) only if the system is stable in the

current state. Since all transitions taken that are triggered by an event (either internal or from the environment) occur in a single RSML microstep, events remain 1 for only one SMV step.

Timing constraints Recall that in Figure 3 there is a timing constraint $t \geq t(\text{entered}(Q)) + 5$ sec, which is equivalent to $t - t(\text{entered}(Q)) \geq 5$ sec. In order to model this constraint, we need the difference between the current time and the time when state Q was last entered. To avoid storing a potentially unbounded value for this difference, we create a variable `Time_Since_Entered_Q` to implement a timer:

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

The assignment says that (case 1) if the machine enters state Q, reset the timer, (case 2) if the machine is stable and the timer is less than 5 seconds, advance the timer and (case 3) otherwise, the timer remains unchanged. Limiting the domains of timers in this way is critical for the efficiency of the SMV translation.

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

Transitions A transition in RSML 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 an SMV 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:

```
DEFINE
  T_S1_S2 :=
    inS1                          -- source state
    & x                           -- trigger event
    & ( (inU & Alt > 1000)        -- guards (col 1)
       | Time_Since_Entered_Q >= 5);  -- (col 2)
```

(Comments in SMV start with "--".) For the most part, the translation of the guards proceeds directly as in the first guard for this example in which `inU` is defined as above and `Alt` is either an SMV variable or defined symbol whose value is compared to the constant 1000. `Time_Since_Entered_Q` is a timer discussed above.

To model the state change for `S`, we have an assignment:

```
ASSIGN
  next(S) :=
    case
      T_S2_S1 | T_S3_S1 : S1;
      T_S1_S2 | T_S3_S2 : S2;
      T_S1_S3 | T_S2_S3 : S3;
      !inS & next(inS) : S1;   -- start state
      1 : S;
    esac;
```

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

Observe that if multiple transitions out of a single state, such as `T_S1_S2` and `T_S1_S3`, are enabled simultaneously (they have the simultaneously fired trigger events and simultaneously satisfiable guarding conditions), then, since SMV always evaluates the conditions in a **case** expression in order, this specifies a deterministic transition in SMV whereas it specifies a nondeterministic transition in RSML. Jaffe et. al. [16] argue that such nondeterministic transitions are usually design flaws in the specification and should be avoided. In Section 5.1 we will describe how to detect undesired nondeterminism in this deterministically modeled specification.

Given these deterministic transitions, output actions (i.e., events) are modeled simply as a logical disjunction of the transitions that generate them. For example:

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

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

Intentionally Nondeterministic Transitions Nondeterministic transitions between different states, such as would be the case if `T_S1_S2` and `T_S1_S3` could be simultaneously enabled, are nearly as easy to model in SMV. In this case the values of `S` and `next(S)` can be used to determine which transition has been taken. For example, we can insert before the first line in the **case** expression above:

```
T_S1_S2 & T_S1_S3 : {S2,S3};
```

The condition states that if the two transitions are enabled simultaneously, the machine will go to S2 or S3 nondeterministically. To generate the correct value for `next(y)` we merely need to replace the disjunct `T_S1_S2`:

```
ASSIGN
  next(y) := (T_S1_S2 & next(inS2)) | T_U1_U2;
```

This method is convenient when the number of nondeterministic options out a single state is small (the most likely reasonable case) but k such options would entail $2^k - k - 1$ additional cases.

A potentially more concise but somewhat more cumbersome translation for this situation, using the TRANS statement of SMV as opposed to the ASSIGN statements used above, can be given as follows:

```
( (T_S1_S2 & next(inS2) & next(y) = 1) |
  (T_S1_S3 & next(inS3) & .. ) |
  (T_S2_S1 & next(inS1) & .. ) |
  (!T_S1_S2 & !T_S2_S3 & !T_S2_S1 & next(S)=S))
& (!(T_S1_S2 & next(inS2))
  & !T_U1_U2 & next(y) = 0 )
```

The default case for state `S` would be included in all transitions that enter state `Q`.

In the unlikely case that a reasonable design includes parallel transitions between the same two states that can be triggered by simultaneously fired triggering events and have simultaneously satisfiable guarding conditions but generate different output actions, it is necessary to enlarge the SMV variable space by including a variable to record which of

the parallel transitions will be taken. Corbett [9, p. 178] gives details of a similar translation, although his translation would use similar variables for all states, not just those with parallel transitions.

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 SMV defined symbols, which do not add variables to the BDD representations. 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.

Comparison with Statecharts In contrast to RSML step semantics, Statechart step semantics (as defined by Pnueli and Shalev [19]) build a set T of transitions that will fire in a step by iteratively computing a closure based on the enabled transitions at the start of the step. Only after the closure is computed do the transitions fire. This appears to be less efficient to model in SMV since one would seem to need an extra boolean variable for each transition in order to record whether or not it is in the set T computed during each step.

4 Obstacles to Model Checking TCAS II 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 II specification feasible.

4.1 TCAS II

TCAS II is an airborne collision avoidance system required on most commercial aircraft. 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 possibly escape maneuvers (resolution advisories) in the vertical direction to the pilot to avoid collision. Examples of resolution advisories (RAs) include Climb, Descend, Increase-Climb ("increase the current climb rate"), Increase-Descend, Climb-VSL0 ("do not descend"), Climb-VSL500 ("do not descend more than 500 ft/min"), etc.

The specification of TCAS II, a 400 page document, was written in RSML. The first obstacle to analyzing the specification was its sheer size. As a first attempt we decided to try to model check 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 part of TCAS called Other-Aircraft, which tracks the state of other aircraft in the vicinity and possibly generates RAs. Up to 30 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. Figure 5 shows the state Composite-RA, one of the twelve parallel substates of Own-Aircraft.

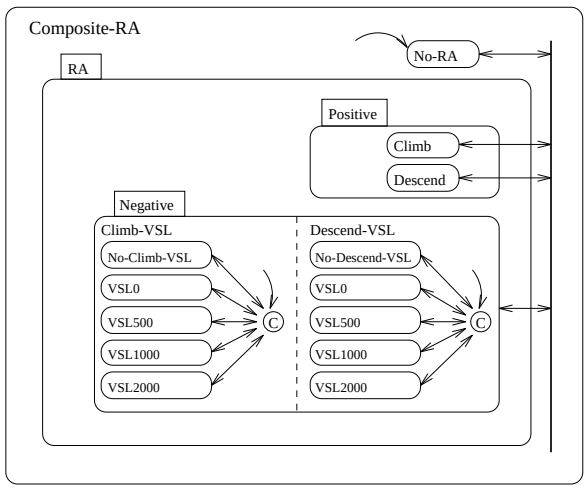

Figure 5: Composite-RA in Own-Aircraft

We treated Other-Aircraft as part of the environment of Own-Aircraft. That is, we created variables for any states of Other-Aircraft that are referenced within Own-Aircraft. Like environment variables, their values were nondeterministic, except that we restricted when these variables could change to ensure correct synchronization. We focused on resolution maneuvers with one intruder aircraft 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 under any variable ordering [3, 20] 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 input variables that we already modeled nondeterministically. So we made the conservative simplification 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.) These simplifications did not cause problems for the properties that we checked and report in Section 5.

4.3 SMV[1]

The performance of BDD-based algorithms is directly related to the size of the BDDs. 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 dynamic variable reordering and conjunctive partitioning [6], which are supported by SMV. 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

[1] This section refers to SMV Release 2.4.4 which was the most recent version to which we had access.

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 to analyze the properties. The result column indicates whether the property was true. The (user + system) time, the number of BDD nodes and the memory allocated were reported by SMV. These include the resources used to construct the transition relation, evaluate the formula and find a counterexample (if the formula was evaluated false). The first row in the table tells the resources used just to build the transition relation. The experiments were performed on a lightly loaded Sun SPARCstation 10 running SunOS 4.1.3 with 128 MB of main memory.

altitude rates. Take altitudes for an example. The specification states that some altitude variables have granularity as fine as "1 to 10 feet." The ranges of some altitude variables are not specified, but they are compared to constants whose values range from 400 feet to 30500 feet. Therefore at least 13 to 15 bits are needed to represent altitudes. However, we found that with these values we could not get the model checker to build the BDDs in a reasonable amount of time.

Initially we got around the problem by redefining the constants so that they fitted in a small range, for example, from 0 to 15 for altitudes and -4 to 3 for altitude rates. (Increasing the numbers 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 formulae, 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 caused some formulae that are false for the specification to evaluate to true for the model.

We could not leave the results of addition and comparison nondeterministic as we did with multiplication and division in Section 4.2, 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.

We eventually realized that the problem with the ranges was due to the variable ordering for the BDDs that SMV was using to represent integer addition and comparison. The BDD for any bit of 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. Although in principle the former may be a better long term solution, the latter method seemed a simpler approach and we were able to use it with great success. We wrote some simple awk scripts for preprocessing the SMV program to allow parameterized macro expansion, loop unrolling, etc. Using these facilities, we implemented efficient addition and comparison in the SMV program and manipulated all the integer variables and constants at the bit level. We can now model the altitudes and altitude rates with the precisions required by the specification.

Another performance problem was that generating a counterexample often took hours even though the formula was determined false within minutes. Evaluating the formula and finding a counterexample (in case the formula was false) were done by the model checker as two separate searches in the reachability graph. For example, to check for an invariant property with the formula AG p (i.e. p is true in all the reachable states), the model checker started from the set of "bad" states (in which p is false), and searched the set of states that could reach the "bad" states by iteratively applying the backward transition relation. 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 a formula representing an invariant property is evaluated false, a counterexample can now be found almost instantly.

5 Results of Model Checking TCAS II

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 5.1, 5.2, 5.3 and Section 5.6) and domain-specific properties (Sections 5.4 and 5.5).

Table 1 reports the resources needed to analyze the properties. The BDD representation of the SMV program has 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[2] is at least 9.6×10^{56}.

[2]We obtained this lower bound by executing SMV with the com-

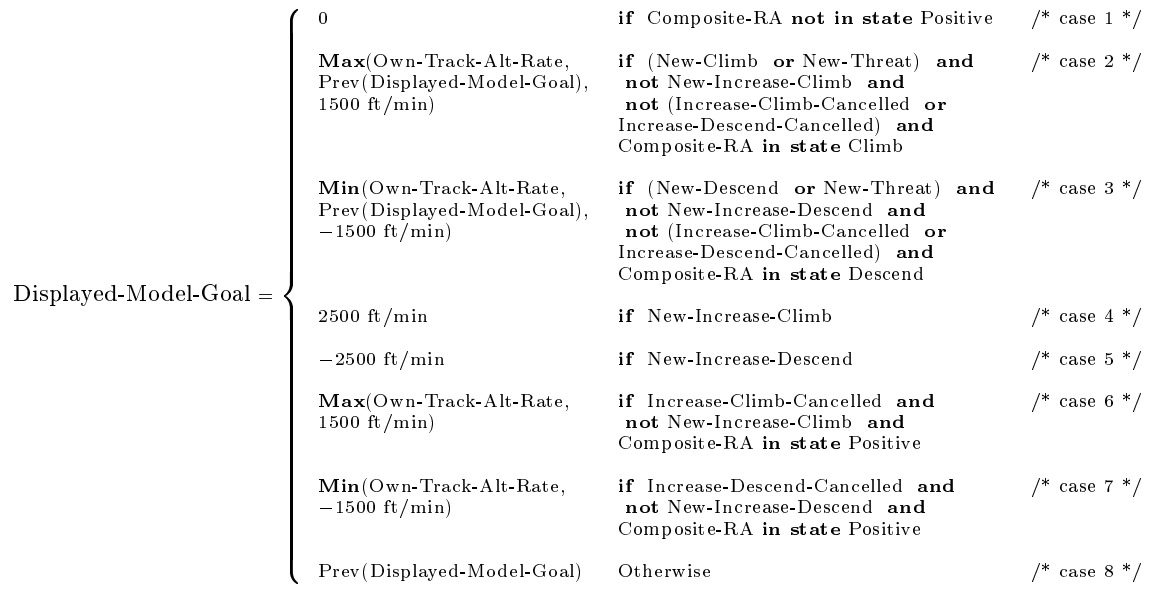

Figure 6: Definition of Displayed-Model-Goal in the TCAS specification. Most of the identifiers are RSML macros or abbreviations, the definitions of which are omitted here due to limited space. (Their truth values depend on Composite-RA and Other-Aircraft.)

5.1 Transition Consistency

There are known nondeterministic transitions in earlier versions of the TCAS specification. So, our first attempt 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 [11], in which there is no unintentional nondeterminism.) These nondeterministic transitions had previously been identified by Heimdahl and Leveson [13] using a different technique. We were interested in checking these properties to verify that model checking could match previous results. In Section 6 we will summarize the differences between our model checking approach and the technique used by Heimdahl and Leveson.

In our example in Figure 2, there are possible nondeterministic transitions from state S. For example, the transitions from S1 to S2 and from S1 to S3 would be enabled 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 enabled; similarly for `T_S1_S3`. So the CTL formula specifies that the two transitions are never enabled simultaneously. Applying this technique to all the states, the model checker was able to find the nondeterministic transitions in that version of the specification.

mand line option -f but without running it to completion. This option forces SMV to find the reachable state space before evaluating any formula.

5.2 Function Consistency

Displayed-Model-Goal, shown in Figure 6, is a function whose value is displayed to the pilot. It 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 is similar to checking for nondeterminism. We defined a boolean symbol `Case-1` for the first Case, and `Case-2` for the second case, and so on, and checked 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 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.

5.3 Step Termination

A step in an RSML state machine may not terminate if the machine contains a cycle of events under the transition relation. However, usually the events in an RSML specification, such as the TCAS specification, form a partial ordering under the transition relation, so it is easy to see that the state machine will always terminate. Alternatively, in our framework we can check for termination with the CTL formula:

```
AG (!Stable -> AF Stable)
```

which states that whenever the state machine is not stable, it will always become stable eventually. This formula was evaluated true for our model of the TCAS specification, as expected.

5.4 Inhibition of Resolution Advisories

A TCAS document [10] states 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 similar to the following:[3]

```
AG ((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. The counterexample it gave revealed a typographical error in a guarding condition in the specification ($>$ instead of $\leq$).[4] 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.

5.5 Output Agreement

In addition to the value of Displayed-Model-Goal, the state of Composite-RA in Figure 5 is also shown to the pilot. 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 that when Composite-RA is Climb, Displayed-Model-Goal could be negative. The CTL formula we checked was roughly:

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

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

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 of the definition of Displayed-Model-Goal, it is ≤ -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 and Displayed-Model-Goal becomes ≤ -1500 ft/min, resulting in contradictory outputs.

5.6 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 in which a variable is referenced before it is defined, e.g., referring to PREV() in the first step.

5.7 Discussion

As shown in Section 5.2, the model checker sometimes found incorrect counterexamples due to the simplifications of the system that we made. It may seem that the repeated process of getting an incorrect counterexample and eliminating 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 have specified Own-Aircraft first and have left Other-Aircraft nondeterministic. Then an analyst could have model checked Own-Aircraft 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, 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.

[3] The actual formulae differ slightly due to some implementation details.

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

6 Related Work

Sreemani and Atlee [21], in work independent of ours, analyzed the A-7E aircraft software requirement specification with SMV, and were also able to successfully check several temporal 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, 14], which does not contain features such as hierarchical states and does not make assumptions like the synchrony hypothesis. In addition, the environment of the A-7E specification is abstracted as a set of predicates, whereas the inputs to our system include numerical values. Numerical calculation and comparison are abundant in the TCAS specification, and they introduce significant problems in the model checking process.

There are a number of other widely researched approaches to handling the state space explosion problem. Corbett recently classified these techniques into several categories [9]. In contrast to our work, which studies a single data point for a single approach, Corbett compared three approaches, model checking, partial order space state reduction, and inequality necessary conditions, all in the context of detecting deadlock in Ada tasking programs. For deadlock, Corbett observed that "no technique was clearly superior to the others, but rather each excelled on certain kinds of programs [9, p. 179]."

The two translations into SMV that Corbett used differ from ours. One translation represented asynchrony by arbitrary sequential interleavings of transitions, eliminating the parallelism that we exploit. The other translation, which he found less successful, represented asynchrony in parallel using extra variables to indicate which transition was executed in each state machine whereas our translation only requires extra variables where parallel nondeterministic transitions occur between the same two states. Use of our translation may have changed the outcome of Corbett's comparison, but further work is needed to determine which approaches are most effective for checking particular properties on certain classes of systems.

Heimdahl and Leveson [13] took a different approach. They analyzed the TCAS specification without exploring the state space. They deduced global properties of the system by composing results of local analysis. Their technique differ from ours in two ways.

First, the properties that we checked were different. Their concerns were transition consistency and completeness [16], which are domain-independent robustness properties. In Section 5.1 we discussed how we checked for a source of transition inconsistency. (They also discussed other sources of transition inconsistency, which we have not addressed.) Completeness intuitively means that a response is specified for every input; more specifically, it means that the disjunction of the guarding conditions of all the transitions with the same triggering event from a particular state form a tautology. In principle this can also be checked in our framework similar to the way consistency is checked. In general, our approach permits analysis of properties that can be expressed as CTL formulae, and is therefore capable of checking domain-specific properties as well (Sections 5.4 and 5.5).

Second, their tool is more efficient for checking transition consistency and completeness. On the other hand, it sometimes produced many spurious errors due to the predicates involving arithmetics in the AND/OR tables, because the predicates were modeled as independent boolean variables. To eliminate the spurious reports they would have to find out the relationships among the predicates. In contrast, we modeled the numbers directly in the BDDs and interleaved their bits in the binary representation to improve performance. In this way, we were able to handle addition and subtraction. Because we explore the reachable state space we generate fewer spurious errors.

7 Conclusions

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 iteratively, by modeling some components nondeterministically and then refining them, proved to be quite powerful. These are critical steps towards realizing symbolic model checking as an effective tool in the process of analyzing software specifications.

What else is needed to make model checking as ubiquitous for software systems as it is already for hardware systems? This is hard to predict with certainty, but a number of directions seem especially promising.

First, Bryant and Chen [4] introduced the BMD (Binary Moment Diagram), a data structure that, in contrast to BDD's, 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 [7], 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.

Second, automating the translation from RSML to input for SMV (or another model checker) appears to be straightforward. It might be reasonable to develop a model checker that directly accepts languages such as RSML or Statecharts, eliminating the need for any source-level translation at all. This is a good example of a place where model checkers developed specifically for software might have some leverage, since the way in which engineers define the state machines often seems to differ between hardware and software.

Third, 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. More generally, 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 believe that this investigation contributes to an increase in optimism that symbolic model checking can over-

come predicted impediments and thus be successful in the analysis of realistic 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] 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.

[4] 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.

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

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

[7] E. Clarke and X. Zhao. Word 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.

[8] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–63, April 1986.

[9] J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, SE-22(3), March 1996.

[10] Federal Aviation Administration, U.S. Department of Transportation. *Introduction to TCAS II*, March 1990.

[11] Federal Aviation Administration, U.S. Department of Transportation. *TCAS II Collision Avoidance System (CAS) System Requirements Specification, Change 6.00*, March 1993.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *Proceedings of International Conference on Theoretical Aspects of Computer Software*, pages 245–264. Springer-Verlag, September 1991.

[20] 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.

[21] 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.

[22] 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.