IEEE Copyright Notice

Published in: ***Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96),*** June 1996

"Feasibility of Model Checking Software Requirements: A Case Study"

Cite as:

T. Sreemani and J. M. Atlee, "Feasibility of model checking software requirements: a case study," *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, Gaithersburg, MD, USA, 1996, pp. 77-88.

BibTex:

```
@INPROCEEDINGS{507877,
author={T. {Sreemani} and J. M. {Atlee}},
booktitle={Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96},
title={Feasibility of model checking software requirements: a case study},
year={1996},
pages={77-88},
month={June}}
```

DOI: http://dx.doi.org/10.1109/CMPASS.1996.507877

Feasibility of Model Checking Software Requirements: A Case Study

Tirumale Sreemani Joanne M. Atlee*

Department of Computer Science
University of Waterloo, Waterloo, Ontario N2L 3G1, Canada
e-mail: tsreeman@uwaterloo.ca, jmatlee@uwaterloo.ca

Abstract

Model checking is an effective technique for verifying properties of a finite specification. A model checker accepts a specification and a property, and it searches the reachable states to determine if the property is a theorem of the specification. Because model checking examines every state of the specification, it is a more thorough validation technique than testing executable specifications.

However, some researchers question the feasibility of model checking, because the size of a specification's state space grows exponentially with respect to the number of variables in the specification. This paper demonstrates the feasibility of symbolically model checking a non-trivial specification: the software requirements of the A-7E aircraft. The A-7E requirements document lists five properties that the designers manually derived from the requirements. Using McMillan's Symbolic Model Verifier, we were able to verify or find a counterexample to each property in less than 10-15 CPU minutes. In particular, we found that an important safety property did not hold.

1. Introduction

Formal notations have been developed to facilitate the specification of precise and unambiguous software requirements [1, 14, 23, 24]. The use of these notations helps to ensure that the requirements writer carefully considers and documents all appropriate software behavior.

Given a formal specification, a *model checker* can automatically verify that certain properties are theorems of the specification. The reachability graph of the specification serves as a logic model, the properties to be verified are expressed as temporal logic formulae, and the model checker exhaustively searches the reachability graph to determine if the properties are true; if they are, then the properties are theorems of the specification. Because the model checker checks the value of the formulae in every state of the specification, model checking is a more thorough validation technique than testing executable specifications.

The feasibility of model checking has been questioned because the size of the reachability graph that is searched grows exponentially with the number of variables used to specify the requirements. However, a number of advances have been made recently that reduce the size of the search space. Symbolic and partial-order model checkers search sets of reachable states rather than individual states [7, 11, 13, 18]. The size of the search space can be reduced further by exploiting symmetry in the specification [19, 21] or by searching an abstraction that only contains information relevant to the property being checked [20, 33]. Using compositional model checking methodologies [12, 27], one can verify that a property is a theorem of *part* of a specification (e.g., a safety kernel [22, 30]) and infer that the property is theorem of the entire specification.

To test the feasibility of model checking non-trivial specifications, we used McMillan's Symbolic Model Verifier (SMV) [7, 25] to model check the software requirements of the A-7E aircraft [1]. The A-7E requirements were written in the Software Cost Reduction (SCR) requirements notation [1, 16, 17]. The specification consists of three concurrent components, each modeling 6 to 18 modes of operation and reacting to 69 input conditions; the theoretical size of the specification's state space is 1.3×10^{22} states. In addition, the A-7E document lists five properties that the requirements writers manually derived from the requirements. We implemented a program that translates SCR requirements into an equivalent SMV specification. Once in the SMV format, we were able to verify or find counterexamples to all five of the A-7E properties. One of the properties that was found to be violated is an important safety property: *it is possible to fire a weapon at a target's coordinates when the aircraft's knowledge of its own coordinates is known to be inaccurate!*

*This work was supported by the Natural Sciences and Engineering Research Council under grant OGP0155243.

Our paper is organized as follows: the mechanics of symbolically model checking SCR software requirements are described in Section 2. The results of our feasibility study are presented in Section 3; this includes descriptions of the changes made to the A-7E specification to facilitate model checking, the results of the model checking, and the performance of the model checker. Finally, we compare our feasibility results with those of other model-checking empirical studies.

2. Symbolic Model Checking of SCR Software Requirements

The SCR requirements notation was developed by a research group at the Naval Research Laboratory as part of a general Software Cost Reduction project [1, 17]. A complete SCR requirements specification describes interface, behavioral, functional, precision, and timing requirements of a software system to be developed. This paper discusses model checking of behavioral requirements only.

2.1. SCR Behavioral Requirements

The environment of a system is abstracted as a set of predicates on environmental variables, called *environmental conditions* [16]. For example, a thermostat that regulates the temperature of a room might define environmental condition SwitchIsOn to represent predicate [On/Off switch = On], and condition TooCold to represent predicate [ActualTemp $<$ (DesiredTemp -3° C)]. If C is the set of environmental conditions, then a *state* is an interpretation of C, in which each condition in C is assigned a truth value; the system's *state space* is the set of allowable interpretations. A state transition is an ordered pair of states whose interpretations differ in the value of one condition.

The behavior of the system is determined by the current state. However, most systems behave the same in several different states. Thus, the system's state space can be partitioned into sets of equivalent states called *modes of operation* or simply *modes*. For example if thermostat condition SwitchIsOn is false, then the system is idle regardless of the value of condition TooCold.

A *primitive event* is a change in the value of one condition [16]. Primitive events @T(SwitchIsOn) and @F(SwitchIsOn) designate condition SwitchIsOn *becoming true* and *becoming false*, respectively. A *conditioned event* is a primitive event whose occurrence depends on the values of other conditions [16]. Event

@T(SwitchIsOn) WHEN [TooCold]

designates condition SwitchIsOn *becoming true* while condition TooCold *is true*. SwitchIsOn is the event's *triggering event* and TooCold is the event's enabling condition or WHEN *condition*. Since a primitive event is a type of conditioned event (whose WHEN conditions are vacuously true), the term conditioned event will henceforth refer to both primitive and conditioned events.

A conditioned event is modeled by any state transition whose interpretations satisfy the event's triggering event and WHEN conditions: the triggering event must be unsatisfied by the first state's interpretation and satisfied by the second state's interpretation, and the WHEN conditions must be satisfied by the first interpretation [1]. For example, a state transition modeling the above conditioned event must satisfy formula

$$\sim SwitchIsOn \wedge TooCold \wedge n(SwitchIsOn)$$

where n() is the *nextstate* operator, used to designate the conditions that hold in the transition's second state.

A *mode transition* is a set of state transitions that cross a mode boundary. A mode transition is modeled by a set of state transitions whose states' interpretations model the same conditioned event, whose source states are elements of the source mode, and whose destination states are elements of the destination mode.

A *mode class* is a set of modes that partitions the state space and a transition relation on the set of modes. If two or more transitions from the same source mode to different destination modes satisfy the same conditioned event, then the mode class is non-deterministic. A subset of the class's modes are designated possible *initial modes*; the actual initial mode is determined by initial values of the conditions.

Figure 1 is an SCR requirements specification of a simple thermostat. The specification of the mode class's transition relation has a tabular format. The top of the table is labeled with the set of environmental conditions. Each row in the table specifies a mode transition from the mode on the left to the mode on the right. A table entry of "@T" (or "@F") under column C1 represents a triggering event @T(C1) (or @F(C1)); a table entry of "t" (or "f") represents a WHEN condition WHEN[C1] (or WHEN[$\sim$C1]). If the value of a condition C1 does not affect the occurrence of a conditioned event, then the table entry is marked with a hyphen ("–"). For example, the first row in Figure 1 specifies a mode transition from OFF to HEAT due to the occurrence of conditioned event @T(SwitchIsOn) WHEN[TooCold]; it specifies the set of state transitions that satisfy formula

$$Off \wedge \sim SwitchIsOn \wedge TooCold \wedge n(SwitchIsOn) \wedge n(Heat)$$

Below the mode transition table is the specification of the system's initial mode.

[1] SCR semantics [16, 31] propose a continuous-time definition for a conditioned event: given event @T(C1) WHEN [C2], if triggering event @T(C1) occurs at time t, then the conditioned event occurs at time t if and only if there exists a non-empty time interval ϵ such that C2 is true throughout interval $[t-\epsilon, t)$. In order to facilitate model checking, this paper uses a discrete-state version of this definition for all conditioned events.

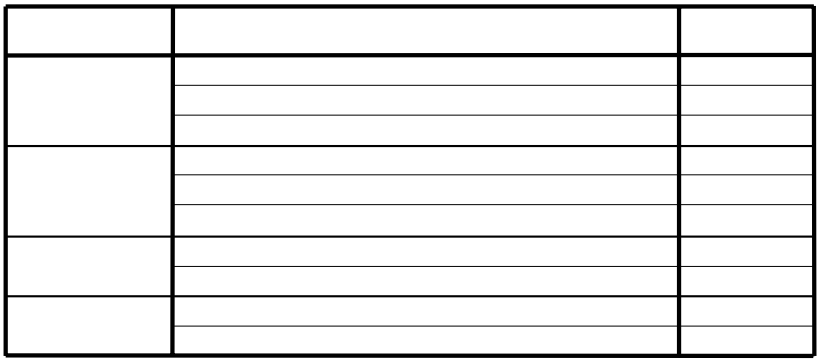

Figure 1. SCR behavioral specification of simple thermostat.

An SCR requirements document contains the specification of one or more mode classes. Each mode class specifies one aspect of the system's behavior, and the system's global behavior is defined to be the composition of the specification's mode classes. At all times, the system is in exactly one mode of each mode class. The transition relation of a set of mode classes is the conjunction of the classes' transition relations.

2.2. Environmental Assumptions

An SCR requirements document also specifies any assumptions of the behavior of the environment. Similar to the NAT relation in Parnas's 4-variable model of system requirements [28], an *assumption* specifies dependencies among the environmental conditions, imposed either by laws of nature or by other mode classes in the system. An example of an environmental assumption is the relationship between thermostat conditions TooCold, TempOk, and TooHot: exactly one of these conditions is true at all times, and the temperature cannot rise (or fall) instantaneously from TooCold to TooHot (or vice versa).

Assumptions are invariant constraints that hold in all states of the specification's reachability graph. Thus, the transition relation of an SCR requirements specification is the conjunction of the specification's mode classes' transition relations with all of the environmental assumptions.

2.3. Translating SCR Requirements into SMV

We have implemented a program that translates an SCR requirements specification into an equivalent SMV specification. Figure 2 contains the SMV representation of the SCR thermostat example depicted in Figure 1.

Environmental conditions, mode classes, and modes are defined as SMV state variables declared in the VAR section. Most conditions are represented as boolean variables. However, if two or more conditions are related such that exactly one condition is true at all times, then the conditions are represented as an enumerated-type variable. For example, each mode class is modeled as an enumerated-type variable, whose value ranges over the names of the mode class's modes.

The system's initial state and transition relation are defined in the ASSIGN section. The initial mode of each mode class is declared using an *init()* statement. In addition, if the system behavior is constrained by the initial value of a variable, then the variable is initialized using the *init()* statement.

The *next()* statement is used to specify the value of a variable in the next state. The transition relation of each mode class is modeled as a *next()* statement that specifies the next value of the mode class variable; because the mode-class variable's next value depends on the variable's current value and the values of the environmental conditions, the *next()* statement uses a *case* expression. Each row in an SCR mode transition table is modeled by a branch in the *case* expression: a boolean expression models the current mode and the transition event, and is followed by the variable's corresponding next value. For example, the first row in the SCR thermostat example is modeled as the following *case* branch

```
Thermostat=Off & !SwitchIsOn & next(SwitchIsOn) &
  Enum1=TooCold : Heat
```

where Thermostat is the mode class variable, Enum1 is a generated name representing the enumeration {TooCold, TempOk, TooHot}, "!" is the SMV *not* connective, and

```
MODULE main
VAR
   Thermostat : {Off, Inactive, Heat, AC};
   Enum1 : {TooCold, TempOk, TooHot};
   SwitchIsOn : boolean;
ASSIGN
   init(Thermostat) := Off;
   init(SwitchIsOn) := 0;
   next(Thermostat) := case
      Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TooCold : Heat;
      Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TempOk : Inactive;
      Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TooHot : AC;
      Thermostat = Inactive & SwitchIsOn & next(!SwitchIsOn) : Off;
      Thermostat = Inactive & SwitchIsOn & !Enum1=TooCold & next(Enum1=TooCold) : Heat;
      Thermostat = Inactive & SwitchIsOn & !Enum1=TooHot & next(Enum1=TooHot) : AC;
      Thermostat = Heat & SwitchIsOn & next(!SwitchIsOn) : Off;
      Thermostat = Heat & SwitchIsOn & !Enum1=TempOk & next(Enum1=TempOk) : Inactive;
      Thermostat = AC & SwitchIsOn & next(!SwitchIsOn) : Off;
      Thermostat = AC & SwitchIsOn & !Enum1=TempOk & next(Enum1=TempOk) : Inactive;
      1 : Thermostat;
   esac;
TRANS
   ( Enum1=TempOK & !next(Enum1=TempOK) & !Enum1=TooHot & next(Enum1=TooHot) ) |
   ( !Enum1=TempOK & next(Enum1=TempOK) & Enum1=TooHot & !next(Enum1=TooHot) ) |
   ( Enum1=Toocold & !next(Enum1=Toocold) & !Enum1=TempOK & next(Enum1=TempOK) ) |
   ( !Enum1=Toocold & next(Enum1=Toocold) & Enum1=TempOK & !next(Enum1=TempOK) ) |
   ( Enum1 = next(Enum1))
TRANS
   ( !(Enum1 = next(Enum1)) & (SwitchIsOn = next(SwitchIsOn))) |
   ( (Enum1= next(Enum1)) & !(SwitchIsOn = next(SwitchIsOn))) |
   ( (Enum1= next(Enum1)) & (SwitchIsOn = next(SwitchIsOn)))
SPEC
   AG((Thermostat=Off)-> !SwitchIsOn)
   AG((Thermostat=Inactive) -> (SwitchIsOn & Enum1=TempOk))
   AG((Thermostat= Heat) -> (SwitchIsOn & Enum1=TooCold))
   AG((Thermostat= AC) -> (SwitchIsOn & Enum1=TooHot))
```

Figure 2. SMV specification of thermostat.

"&" is the SMV *and* connective. If the boolean expression in the above *case* branch is satisfied, then the next value of variable Thermostat is Heat. In general, the next value of a mode class variable is the destination mode in the first *case* branch whose boolean expression evaluates to true. A final *case* branch with boolean expression "1" is equivalent to an *else* clause. Else clauses are used in mode-class transition relations to explicitly indicate that all other state transitions do *not* reflect a mode transition.

Our program translates simple environmental assumptions (e.g., implications) into DNF assertions about the current and next values of the dependent conditions. Each of these expressions is declared in a TRANS section and acts as a constraint on the system's transition relation. In addition, our program can generate a DNF expression that ensures that a state transition reflects a change in the value of only one (independent) environmental condition; this expression is also declared in a TRANS section and acts as a constraint on the transition relation.

Properties to be verified with respect to an SMV specification are declared in the SPEC section.

2.4. Goals to be Model Checked

An SCR requirements specification often includes a set of properties or putative theorems (called *goals*) that should be true if the behavioral requirements have been written correctly. Such properties are redundant information that is included in the specification because the reader might not deduce them from the tabular specifications.

The SMV model checker can verify goals that are expressed as formulae in the Computational Tree Logic (CTL) branching-time temporal logic [8]. SMV evaluates a formula with respect to a particular state in the specification's reachability graph, based on the set of possible execution paths emanating from the state. Because *the* future path of the system's execution is unknown, CTL temporal operators are quantified over the set of possible futures (e.g., a property ϕ is true in *some* next state or in *all* next states).

The syntax and semantics for CTL formulas are defined in [8]. The subset of CTL used in this paper is summarized below:

1. Every propositional variable is a CTL formula.
2. If ϕ and ψ are CTL formulas, then so are: $!\phi$, $\phi \& \psi$, $\phi \mid \psi$, $AX\phi$, $EX\phi$, $EF\phi$, $AG\phi$.

The symbols ! (*not*), & (*and*), and | (*or*) are logical connectives and have their usual meanings. X is the *nextstate* operator, and formula $EX\phi$ ($AX\phi$) is true in state s_i if and only if (iff) formula ϕ is true in some (in every) successor state of s_i in the reachability graph. F is the *future* operator, and $EF\phi$ is true in state s_i iff along some path from s_i there exists a future state in which ϕ holds. Finally, G is the

global operator, and $AG\phi$ is true in state s_i iff ϕ holds in every state along every path emanating from s_i.

The following are properties of the thermostat system, expressed as CTL formulae:

$$AG((Thermostat = Off) \rightarrow !SwitchIsOn)$$
$$AG((Thermostat = Heat) \rightarrow (SwitchIsOn \wedge (Enum1 = TooCold)))$$
$$AG((Thermostat = Inactive) \rightarrow (SwitchIsOn \wedge (Enum1 = TempOk)))$$
$$AG((Thermostat = AC) \rightarrow (SwitchIsOn \wedge (Enum1 = TooHot)))$$

All of the above formulae are properties expressing mode invariants. For example, if the system is in mode HEAT it should be invariantly *true* that the SwitchIsOn and the temperature is TooCold.

2.5. Symbolic Model Checking

The SMV model checker accepts an SMV transition relation and a goal expressed as a CTL formula. The checker then constructs a propositional-logic representation of the goal. The specification satisfies the goal iff the interpretations of the system's initial states satisfy the goal's propositional-logic representation. Otherwise, the checker reports that the goal is not always satisfied, and it presents an execution path that violates the goal.

If the goal is expressed as a propositional logic formula, then the formula representing the set of states satisfying that goal is the goal itself; the specification satisfies the goal iff the interpretations of the initial states satisfy the goal. For example if ϕ and ψ are propositional variables (e.g., SCR environmental conditions), then the specification satisfies goal $\phi \wedge \psi$ iff all of the initial states' interpretations satisfy both ϕ and ψ.

If a goal states a property about the next state, then the model checker uses the logic model's transition relation to build the appropriate formula. For example, if formula $EX\phi$ represents the set of states that the transition relation R maps into ϕ, then formula $R^{-1}(\phi)$ (where R^{-1} is the inverse relation of R) represents the set of states in which goal $EX\phi$ holds. Similarly, formula $!(R^{-1}(!\phi))$ represents the set of states in which goal $AX\phi$ holds.

All of the other CTL modal operations can be defined as fixed points of the EX and AX operators. For example,

$$EF(f) = minY.(f \vee EX(Y))$$

is a fixed point definition of EF. Y_0 is *false* (the empty set of states); Y_1 is the set of states satisfying f (i.e., the set of states that can reach a state satisfying f via 0 applications of the transition relation); Y_2 is the set of states $f \vee EX(f)$ (i.e., the set of states that can reach a state satisfying f via 0 or 1 applications of the transition relation); Y_3 is the set of states $f \vee EX(f) \vee EX(EX(f))$ (i.e., the set of states that can reach a state satisfying f via 0, 1, or 2 applications of the transition relation); etc. The computation terminates when the least fixed point is reached; that is after n iterations, where n is the smallest integer such that $Y_{n-1} = Y_n$.

AG has a greatest fixed point definition:

$$AG(f) = maxY.(f \wedge AX(Y))$$

Y_0 is *true* (the set of all states); Y_1 is the set of states satisfying f (i.e., the set of states for which f is true after 0 applications of the transition relation); Y_2 is the set of states satisfying $f \wedge AX(f)$ (i.e., the set of states for which f is *true* after 0 and 1 applications of the transition relation); Y_3 is the set of states satisfying $f \wedge AX(f) \wedge AX(AX(f))$ (i.e., the set of states for which f is *true* after 0, 1, and 2 applications of the transition relation); etc. The computation terminates when the greatest fixed point is reached; that is after n iterations, where n is the smallest integer such that $Y_{n-1} = Y_n$.

Once a propositional-logic expression PG for goal G has been constructed, the model checker verifies that the specification's set of initial states, represented by formula I, satisfies the predicate:

$$I \rightarrow PG$$

3. Model Checking the A-7E Software Requirements

The A-7E software requirements specification consists of three mode classes, each describing a different aspect of the system's behavior.

The modes in the Navigation/Alignment/Test mode class correspond to different procedures for monitoring and controlling the aircraft's position, velocity, wind data, and alignment (with respect to the earth's East, North, and vertical coordinate system); if the aircraft is on the ground, a series of Tests can be run to detect logic errors in the computer or its input/output interfaces. The modes in the Navigation Update modeclass correspond to different procedures for recalculating the computed position of the aircraft. The modes of the Weapon Delivery modeclass correspond to different procedures for identifying and tracking the positions of targets and for releasing weapons [1].

The A-7E specification documents five constraints on the allowable combinations of modes in different mode classes. These constraints are safety properties that state that the system should not be performing (or should only be performing) certain pairs of functions at the same time. These constraints guided the construction of the tabular requirements. They are included in the introduction to the specification of the mode transition tables to enhance readability of the tables: once it is determined that the system is in one mode of the one of the mode classes, the mode-combination properties can be used to avoid checking whether the system is

in certain modes in the other mode classes. We wanted to verify whether the tables enforced the mode-combination properties, as intended.

3.1. Changes made to the A-7E specification

We needed to make several changes to the A-7E mode transition tables to ensure that the model checking results would be accurate. Most changes involved documenting dependencies among the specification's variables. Other changes were made to bring the specification in line with the syntax and semantics of the SCR requirements notation.

3.1.1. Documenting dependencies

The environmental conditions are not independent variables; that is, there are conditions whose values are constrained by other environmental conditions. Although these environmental assumptions are documented in the A-7E specification, they are scattered throughout the document. Careful (manual) analysis of the specification revealed 17 such environmental assumptions. For example, Guns(GN), Rockets(RK), Walleye(WL), Special(SP), Shrike(SK) and Uncataloged(UN) are weapon classes used in the Weapon Delivery mode class. However, in addition to those mentioned above, the requirements document also refers to weapon classes High Dual, Medium Dual, etc. We defined weapon class as an enumeration of {GN, RK, WL, SP, SK, UN, Other}, where 'Other' abstractly represents weapon class values that are not important to the mode transition specification.

3.1.2. Correcting syntax and explicating semantics

The A-7E specifiers had deviated from their own specification notation to make the tables shorter and more readable. Here, we describe the changes we made to bring the specification into conformance with SCR syntax and semantics.

Added condition describing mode class value. The mode transition table for Navigation Update modeclass is affected by "environmental condition" Weapon mode=BOC. This condition is true exactly when the WeaponDelivery modeclass is in mode BOC. To enforce this constraint, we replaced all occurrences of the condition with predicate WpnDel=BOC, where the variable WpnDel records the current mode of the Weapon Delivery mode class.

Accommodated transitions due to mode entry. In the Weapon Delivery mode class, there are two modes, OFF-MFS and WD-MFS, whose transitions out of the respective modes occur upon mode entry. For example, all of the transitions leaving mode OFF-MFS are activated by @T(In OFF-MFS). Thus the system never remains in either of these two modes. It appears that this technique is intended to model hierarchical modes, where modes OFF-MFS and WD-MFS are abstractions of sets of component modes. Entry into the parent mode is actually an entry into one of the component modes, depending on the event causing the mode entry.

However, transitions activated by mode entry events can lead to a logical inconsistency. Because the transitions entering and leaving modes OFF-MFS and WD-MFS occur simultaneously, two modes of the same mode class (the two destination modes of the simultaneous transitions) are true at the same time. This contradicts the constraint that the system is in exactly one mode of each mode class at any time. We could have manually eliminated the intermediate modes OFF-MFS and WD-MFS, thereby flattening the hierarchical mode structure. To do this, we would have had to find all transitions entering modes OFF-MFS and WD-MFS and replace each such transition with the set of transitions leaving modes OFF-MFS and WD-MFS, respectively; the transition event of the new transition would be a combination of the triggering event and WHEN conditions of the component transitions. However, we were not confident that we could accurately automate this transformation, and doing it manually would have been time consuming and error prone.

Instead, we added a new condition *PWpnDel* to the A-7E specification which records the value of the Weapon Delivery mode class in the previous state; and we replaced all triggering events @T(In OFF-MFS) and @T(In WD-MFS) with @T(PWpnDel=OFF-MFS) and @T(PWpnDel=WD-MFS), respectively.

Combined two mode transition tables into one. The events that cause transitions between the modes in the Align/Nav/Test mode classes are described using two tables. The first table describes the mode transitions while the aircraft is on ground (when condition (ACAIRB=Yes) is false), and the second table describes transitions when the aircraft is airborne (when condition (ACAIRB=Yes) is true). To facilitate model checking the mode class as a whole, we joined the two tables. To retain the known value of condition (ACAIRB=Yes), we added WHEN condition [!(ACAIRB=Yes)] to all transition events from the first table in which (ACAIRB=Yes)'s value was unspecified, and we added WHEN condition [(ACAIRB=Yes)] to all transition events from the second table in which (ACAIRB=Yes)'s value was unspecified.

Added single event constraint. The A-7E mode transition tables were written with the assumption that only one monitored variable can change its value at a time. This assumption was made because it was known that in the system's implementation, changes to monitored variables would be

queued and events would be seen by the system one at a time, even if the events occurred simultaneously. To enforce this constraint, our program generates a DNF expression that states only one variable can change its value at any time. However, the added constraint does allow dependent conditions to change value simultaneously [3].

3.2. Verifying A-7E properties

We model checked the five mode-combination properties against the 1988 version of the A-7E mode transition tables.

The first property states that the system must be in one and only one of the following modes:

Lautocal	DIG	Grtest
Sautocal	DI	
Landaln	I	
SINSaln	UDI	
01Update	OLB	
HUDaln	Mag sl	
Airaln	Grid	
	IMS fail	
	PolarDI	
	PolarI	

In the 1978 version of the specification, these modes were distributed among three modeclasses: Navigation, Alignment and Test. The transitions between these modes were so interrelated that the modes were combined into a single modeclass (the Align/Nav/Test modeclass) in the 1988 version of the A-7E specification. Because the system is always in exactly one mode of every modeclass at all times, the above property holds. The CTL representation of this property is

$$
\begin{aligned}
&AG\,(\\
&(ANT{=}Lautocal \wedge\ !(ANT{=}Sautocal \vee ANT{=}Landaln \vee \\
&ANT{=}SINSaln \vee ANT{=}Update \vee ANT{=}HUDaln \vee \\
&ANT{=}Airaln \vee ANT{=}DIG \vee ANT{=}DI \vee ANT{=}I \vee \\
&ANT{=}UDI \vee ANT{=}OLB \vee ANT{=}Mag\text{-}sl \vee ANT{=}Grid \vee \\
&ANT{=}IMS\text{-}fail \vee ANT{=}PolarDI \vee ANT{=}PolarI \vee \\
&ANT{=}Grtest)) \\
&\bigvee \\
&(ANT{=}Sautocal \wedge\ !(ANT{=}Lautocal \vee ANT{=}Landaln \vee \\
&ANT{=}SINSaln \vee ANT{=}Update \vee ANT{=}HUDaln \vee \\
&ANT{=}Airaln \vee ANT{=}DIG \vee ANT{=}DI \vee ANT{=}I \vee \\
&ANT{=}UDI \vee ANT{=}OLB \vee ANT{=}Mag\text{-}sl \vee ANT{=}Grid \vee \\
&ANT{=}IMS\text{-}fail \vee ANT{=}PolarDI \vee ANT{=}PolarI \vee \\
&ANT{=}Grtest)) \\
&\bigvee \\
&\ldots)
\end{aligned}
$$

where *ANT* is the variable for the Align/Nav/Test modeclass. We verified the property to be true.

The second property states that the system may be in at most one of the following modes at a time:

HUDUpd	Nattack	Walleye
RadarUpd	Noffset	Snattack
MapUpd	BOC	Snoffset
TacUpd	BOCFlyto0	SBOC
FlyUpd	BOCoffset	SBOCFlyto0
	CCIP	SBOCoffset
	A/G guns	HUDdown1
	HUDdown2	

This property states that certain combinations of Navigation Update modes and Weapon Delivery mode are not possible. When the aircraft is in one of the listed Navigation Update modes, it is updating the calculated coordinates of the aircraft. The aircraft must not be calculating the location of the weapon targets while it is recalculating the coordinates of the aircraft. The CTL formula representing this property is

$$
\begin{aligned}
&AG(\\
&(NavUpd{=}HUDUpd \wedge\ !(WpnDel{=}Nattack \vee \\
&WpnDel{=}MdWalleye \vee WpnDel{=}Noffset \vee \\
&WpnDel{=}Snattack \vee WpnDel{=}BOC \vee WpnDel{=}Snoffset \\
&\vee WpnDel{=}BOCFlyto0 \vee WpnDel{=}SBOC \vee \\
&WpnDel{=}BOCoffset \vee WpnDel{=}SBOCFlyto0 \vee \\
&WpnDel{=}CCIP \vee WpnDel{=}SBOCoffset \vee \\
&WpnDel{=}AG\text{-}Guns \vee WpnDel{=}HUDdown1 \vee \\
&WpnDel{=}HUDdown2)) \\
&\bigvee \\
&(NavUpd{=}RadarUpd \wedge\ !(WpnDel{=}Nattack \vee \\
&WpnDel{=}MdWalleye \vee WpnDel{=}Noffset \vee \\
&WpnDel{=}Snattack \vee WpnDel{=}BOC \vee WpnDel{=}Snoffset \\
&\vee WpnDel{=}BOCFlyto0 \vee WpnDel{=}SBOC \vee \\
&WpnDel{=}BOCoffset \vee WpnDel{=}SBOCFlyto0 \vee \\
&WpnDel{=}CCIP \vee WpnDel{=}SBOCoffset \vee \\
&WpnDel{=}AG\text{-}Guns \vee WpnDel{=}HUDdown1 \vee \\
&WpnDel{=}HUDdown2)) \\
&\bigvee \\
&\ldots \\
&\bigvee \\
&(!(NavUpd{=}HUDUpd \vee NavUpd{=}RadarUpd \vee \\
&NavUpd{=}MapUpd \vee NavUpd{=}TacUpd \vee \\
&NavUpd{=}FlyUpd \vee WpnDel{=}Nattack \vee \\
&WpnDel{=}MdWalleye \vee WpnDel{=}Noffset \vee \\
&WpnDel{=}Snattack \vee WpnDel{=}BOC \vee WpnDel{=}Snoffset \\
&\vee WpnDel{=}BOCFlyto0 \vee WpnDel{=}SBOC \vee \\
&WpnDel{=}BOCoffset \vee WpnDel{=}SBOCFlyto0 \vee \\
&WpnDel{=}CCIP \vee WpnDel{=}SBOCoffset \vee \\
&WpnDel{=}AG\text{-}Guns \vee WpnDel{=}HUDdown1 \vee \\
&WpnDel{=}HUDdown2)))
\end{aligned}
$$

where NavUpd and WpnDel are the variables for the Navigation Update and Weapon Delivery modeclasses, respec-

tively. Initially, the model checker determined that the property was false. While analyzing the results, we found that we had not modeled the events triggered by mode entry events(@T(In WD-MFS) and @T(In OFF-MFS)) properly: we had not enforced the constraint that no other event could occur while transitions activated by these mode entry events occurred. We corrected this and found that the property was true.

The third property states that the system may only be in Navigation Update mode AflyUpd when it is in either Weapon Delivery mode BOC or mode SBOC. In Navigation Update modes, the aircraft position is updated using the coordinates of a reference point on the ground. The pilot designates when the aircraft is directly above the reference point. The system then usually displays both the newly calculated position (based on the coordinates of the reference point) and the previously used position (computed by the aircraft's Operational Flight Program), and the pilot can decide which value should be used as the updated position of the aircraft. In AflyUpd, the two positions are not displayed. Instead, the position of the aircraft is updated based on the coordinates of the reference point. This kind of update is only allowed when the system is in mode BOC (or SBOC), because in BOC the weapon target is the reference. Thus, the coordinates of the target are taken as the correct position of the aircraft and the target is bombed.

The CTL formula representing this property is

$$AG(NavUpd{=}AflyUpd \rightarrow (WpnDel{=}BOC \vee WpnDel{=}SBOC))$$

The model checker determined that this property was true.

The fourth property states that the system can be in AA-Guns and Manrip together or separately:

$$EF(WpnDel{=}AA\text{-}Guns) \wedge EF(WpnDel{=}Manrip) \wedge EF(WpnDel{=}AA\text{-}Guns \wedge WpnDel{=}Manrip)$$

This formula was determined to be false. This result was not surprising since AA-Guns and Manrip are modes in the same modeclass and a modeclass can be in only of its mode at any time.

The fifth property states that the system may not be in any of the following Weapon Delivery modes if the Navigation mode is IMS fail, Mag sl or Grid and the condition ADC Up is false:

Nattack	Noffset	BOC
BOCFlyto0	BOCoffset	CCIP
HUDdown1	HUDdown2	Walleye
Snattack	Snoffset	SBOC
SBOCFlyto0	SBOCoffset	

When ADC Up is false, the instruments that provide data to calculate the aircraft's horizontal velocity are dislaying "unreasonable" values. Consequently, the horizontal velocity is not recalculated and the program uses the stale value. Under these circumstances, the system should not be in a Weapon Delivery mode that uses the aircrafts horizontal velocity to calculate the weapon target's location. The CTL formula representing this property is

$$AG(((ANT{=}Mag\text{-}sl \vee ANT{=}Grid \vee ANT{=}IMS\text{-}fail) \wedge !ADCUp) \rightarrow !(WpnDel{=}BOC \vee WpnDel{=}SBOC \vee WpnDel{=}Snattack \vee WpnDel{=}Snoffset \vee WpnDel{=}Nattack \vee WpnDel{=}Noffset \vee WpnDel{=}BOCFlyto0 \vee WpnDel{=}SBOCFlyto0 \vee WpnDel{=}HUDdown1 \vee WpnDel{=}HUDdown2))$$

The condition ADC Up does not affect any mode transition nor does its value depend on the value of the modeclasses. Therefore, we model checked the property without considering the value of condition ADC Up. The modified property was found to be false.

3.3. Feasibility of Symbolic Model Checking

How large is the reachability graph of the A-7E aircraft? The specification consists of three mode classes, each composed of 6 to 18 modes for a total of 41 modes. There are 69 environmental conditions that affect the mode transition tables. In addition, we added 7 new environmental conditions when we modified the specification for verification. The SMV model of the system consists of 54 boolean and 9 enumerated variables; the theoretical size of the state space is 1.3 x 10^{22} states[2]. This section describes the memory resources and execution time needed to symbolically model check the A-7E specification.

3.3.1. Memory Resources

SMV uses Binary Decision Diagrams (BDDs) to represent boolean functions. A BDD is a directed acyclic graph which represents a boolean formula [6]. For example, the BDD in Figure 3(a) represents the boolean formula (a $\wedge$ b) $\vee$ (c $\wedge$ d). One can determine the value of the formula for any assignment of variable values by traversing the tree from root to leaf; at each node, the branch taken depends on the value of the variable labeled in the node. For example, the assignment of values (0, 0, 1, 1) to variables (a, b, c, d) results in a traversal that leads to the leaf labeled 1. Hence, this assignment satisfies the formula.

[2] The theoretical size of the state space is calculated by taking the product of the numbers of possible variable values. For example, if a specification only declares two variables, a boolean and an enumerated type Enum : {one, two, three}, then the size of the state space is 2 x 3 = 6 states. Note that the above calculation does not inflate the state space size by including value combinations prohibited by enumerations; it does, however, include value combinations prohibited by the other declared environmental assumptions.

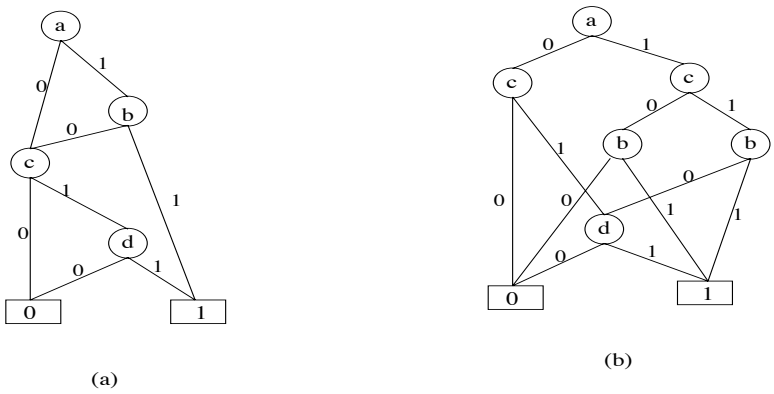

Figure 3. BDDs with different variable ordering.

When model checking SCR requirements, SMV generates one BDD that represents both the transition relations of the mode classes and the constraints specified in the TRANS section. The structure of the generated BDD is sensitive to the order in which variables are declared in the specification's VAR section: the BDD is constructed so that variables in the BDD are traversed in the order in which they are declared. For example, the BDD in Figure 3(a) is constructed from variable ordering a - b - c - d, whereas the BDD in Figure 3(b) is constructed from variable ordering a - c - b - d.

Different orderings of variable declarations can have an enormous effect on the size of a function's BDD representation. In one of our earliest attempts to model check pieces of the A-7E specification, SMV could not construct the BDD representation of the transition relation of a single mode-class (the Navigation Update mode class) after executing for two weeks![3] When we reordered the variable declarations, the mode class's transition relation was generated in about 15 CPU seconds.

When model checking the complete A-7E specification, we tried approximately 10-15 different orderings of each mode class's variables. In the end, the size of the transition relation that was model checked consisted of 64,490 BDD nodes; a total of 262,863 BDD nodes and 5.4 Megabytes were allocated to construct the transition relation.

3.3.2. CPU Resources

Table 1 indicates how long it took to verify each of the five mode-combination properties. The timing information shown for properties 1-3 consists of the time it took to build the transition relation plus the time it took to verify the respective property. For properties 4 and 5, the timing information also includes the time it took to find and display a counterexample of the false formula.

[3] This exercise was performed on a lightly-loaded DEC Alpha, with 256MB of RAM and over 1GB of swap space.

Property	User time (CPU sec)	System time (CPU sec)	Result
1	441.814	18.833	True
2	734.183	32.683	True
3	592.583	28.733	True
4	1065.63*	21.483*	False
5	780.233*	36.433*	False

* Includes time to generate counter-example.

Table 1. Execution times of verifications.

When model checking property 4, SMV attempts to find a counter-example of the false formula. However, property 4 is a statement about the reachability of a particular state. How can the counterexample facility demonstrate that a state is in fact unreachable? It produces the set of states that *can* reach the particular state in question. A considerable amount of time is spent producing this counterexample.

All five exercises were performed on a lightly-loaded SUN with 128MB of RAM and 410MB of swap space.

4. Related Work

Early model checkers (such as EMC [8], MCB [4], the Concurrency Workbench [9], and MEC [2]) verify properties by constructing the specification's reachability graph and then evaluating the property in every reachable state. Because the size of a specification's reachability graph grows exponentially with respect to the number of variables and parallel components comprising the specification, these tools have been used primarily to check small communication protocols [9, 29], mutual exclusion algorithms [2, 32], and small synchronous [5] and asynchronous [10] circuits.

With the advent of symbolic model checking, it has been possible to model check larger systems. For example, the consistency of a distributed cache in the Gigamax multiprocessor, with a potential state space of 10^{13} states, was veri-

fied in "a few minutes" using the SMV model checker [26]. SMV has also been used to detect deadlock in the Hewlett-Packard's Summit bus converter chip, which is responsible for communication between a high-speed processor bus and a low-speed I/O bus [15]; it took 8 to 30 minutes to verify properties of the converter, whose specification consisted of 195 state variables, resulting in a potential state space of 10^{58} states. However, these case studies involved hardware systems, and symbolic model checking is known to exploit the regular structure inherent in digital circuits.

SMV has also been used to verify the IEEE 802.3 Ethernet CSMA/CD communication protocol which allocates a shared channel for multiple users. Properties verified include collision detection and correct data transmission. The number of reachable states in the model was 10^3 to 7.5×10^6 depending on the number of transmitters and the type of model used: asynchronous or synchronous. The model used here has repetitive elements, including multiple instances of receivers or transmitters. The complexity and size of the model can be reduced by taking away the repetitive elements, whereas in the A-7E system there are no repetitive elements and the system as a whole has to be model checked.

Wing and Vaziri-Farahani [33] have verified abstract models of three cache coherence protocols used in distributed file systems. SMV took less than a second to check over 43,600 reachable states. Abstractions were obtained based on the formula being verified or by exploiting domain- or application-specific knowledge [33]. These are manually obtained abstractions that require a lot of case-specific knowledge and human thinking. In contrast, the SCR methodology requires the requirements writer to create abstractions of the (potentially infinite) environmental state space by determining which predicates on values of environmental variables affect mode transitions. Thus, the use of the SCR notation provides an easy method of obtaining abstractions.

5. Conclusion

The use of formal notations helps the requirements writer ensure that all cases of appropriate system behavior are considered and documented. Automated techniques to analyze requirements documents would make it easier to write and review the documents, as well as encourage other requirements designers to adopt the use of formal methods.

One of the advantages of model checking over other automated validation and verification techniques is the degree to which the verification is automated. Simulators execute previously composed test cases, but a human must decide whether or not the result of each test case is correct. Theorem provers are interactive; they vary as to which proof techniques are automated, but they all require some human input that specifies when to apply certain inference rules and proof techniques.

The question most asked about model checking is whether and how the technique scales. This and other case studies clearly demonstrate that model checking reasonably large specifications is feasible.

Acknowledgements

We would like to thank researchers in the Information Technology Division at the Naval Research Laboratory for providing us with an on-line version of the A-7E software requirements document. In particular, discussions with Connie Heitmeyer raised and resolved many questions about the semantics of SCR requirements.

We would also like to thank Dave Parnas and other members of the Communications Research Laboratory at McMaster University for their comments on a presentation of this work. In addition, Dave Parnas's and Paul Clements's help in interpreting informal comments in the A-7E document and in recalling the original requirements writer's motivations and design decisions was invaluable.

References

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

[2] A. Arnold. "MEC: a system for constructing and analysing transition systems". In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 117–132, June 1989.

[3] J. Atlee and A. Malton. "The Non-simultaneity of SCR Events". Presented at the Fifth International SCR Workshop, February 1996.

[4] M. Browne. *Automatic Verification of Finite State Machines Using Temporal Logic*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1989.

[5] M. Browne, E. Clarke, and D. Dill. "Automatic Verification of Sequential Circuits Using Temporal Logic". *IEEE Transactions on Computers*, C-35(12):1035–1044, December 1986.

[6] R. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[8] E. Clarke, E. Emerson, and A. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[9] R. Cleaveland, J. Parrow, and B. Steffen. "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems". *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[10] D. Dill and E. Clarke. "Automatic verification of asynchronous circuits using temporal logic". *IEE Proceedings*, 133(5):276–282, September 1986.

[11] D. Dill, A. Drexler, A. Hu, and C. Yang. "Protocol Verification as a Hardware Design Aid". In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[12] K.L. McMillan E.M. Clarke, D.E. Long. "Compositional Model Checking". In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, June 1989.

[13] P. Godefroid and P. Wolper. "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties". In *Proceedings of the Third International Workshop on Computer Aided Verification, Lecture Notes in Computer Science 575*, pages 332–342, July 1991.

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

[15] C. Harkness and E. Wolf. "Verifying the Summit Bus Converter Protocols with Symbolic Model Checking". *Formal Methods in System Design*, 4(2):83–97, February 1994.

[16] C. Heitmeyer, B. Labaw, and D. Kiskis. "Consistency Checking of SCR-Style Requirements Specifications". In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, pages 56–65, March 1995.

[17] 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.

[18] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic model checking for real-time systems". In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science*, pages 394–406. IEEE Computer Society Press, 1992.

[19] C.N. Ip and D.L. Dill. "Better Verification Through Symmetry". *Formal Methods in System Design*, (to appear).

[20] D. Jackson. "Abstract model checking of infinite specifications". In *Proceedings of Formal Methods Europe Conference*, 1994.

[21] D. Jackson. "Exploiting Symmetry in the Model Checking of Relational Specifications". Technical Report CMU-CS-94-219, School of Computer Science, Carnegie-Mellon University, 1994.

[22] N. Leveson. "Design for safe software". In *Proceedings of the American Institute of Aeronautics and Astronautics 21st Aerospace Sciences Meeting*, January 1983.

[23] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. "Requirements Specification for Process-Control Systems". *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[24] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Spinger-Verlag, 1992.

[25] K. McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[26] K.L. McMillan and J. Schwalbe. "Formal Verification of the Gigamax Cache Consistency Protocol". In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–251, 1991.

[27] J. Ostroff. Specifying and Verifying Real-Time Reactive Systems in TTM/RTTL. Technical Report CS-ETR-94-08, Department of Computer Science, York University, 1994.

[28] D. Parnas and J. Madey. Functional Documentation for Computer Systems Engineering (Version 2). Technical Report CRL Report 237, Department of Electrical and Computer Engineering, McMaster University, 1991.

[29] J. Parrow. "Verifying a CSMA/CD Protocol with CCS". In *Proceedings of the Eighth IFIP Symposium on Protocol Specification, Testing, and Verification*, pages 373–387, June 1988.

[30] J. Rushby. "Kernels for Safety?". In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989.

[31] J. van Schouwen. The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems. Technical Report TR-90-276, Department of Computing and Information Science, Queen's University, Kingston, Ontario, May 1990.

[32] D.J. Walker. "Analyzing mutual exclusion algorithms using CCS". *Formal Aspects of Computing*, 1:273–292, 1989.

[33] J. Wing and M. Vaziri-Farahani. "Model Checking Software Systems: A Case Study". In *Proceedings of the ACM SIGSOFT Conference on Foundations of Software Engineering*, October 1995.