ELSEVIER

Science of Computer Programming 36 (2000) 65–96

Science of Computer Programming

www.elsevier.nl/locate/scico

requirements: an experience report ☆

Mats P.E. Heimdahl[a,*], Barbara J. Czerny[b]

[a] *Department of Computer Science and Engineering, University of Minnesota, Institute of Technology, 4-192 EE/CS Bldg. Minneapolis, MN 55455, USA*

[b] *Department of Computer Science, Michigan State University, 3115 Engineering Building, East Lansing, MI 48824-1226, USA*

Abstract

In a previous investigation we formally defined procedures for analyzing hierarchical state-based requirements specifications for two properties: (1) completeness with respect to a set of criteria related to robustness (a response is specified for every possible input and input sequence) and (2) consistency (the specification is free from conflicting requirements and undesired non-determinism). Informally, the analysis involves determining if large Boolean expressions are tautologies. We implemented the analysis procedures in a prototype tool and evaluated their effectiveness and efficiency on a large real world requirements specification expressed in an hierarchical state-based language called Requirements State Machine Language. Although our initial approach was largely successful, there were some drawbacks with the original tools. In our initial implementation we abstracted all formulas to propositional logic. Unfortunately, since we are manipulating the formulas without interpreting any of the functions in the individual predicates, the abstraction can lead to large numbers of spurious (or false) error reports. To increase the accuracy of our analysis we have continually refined our tool with decision procedures and, finally, come to the conclusion that theorem proving is often needed to avoid large numbers of spurious error reports. This paper discusses the problems with spurious error reports and describes our experiences analyzing a large commercial avionics system for completeness and consistency.

Keywords: Software specification verification; Static analysis; Theorem proving; PVS; State-based models; Completeness; Consistency; RSML

1. Introduction

Languages based on finite state machines, for example, Statecharts [14–16], Software Cost Reduction (SCR) [19, 21], and the Requirements State Machine Language (RSML) [23], are powerful modeling languages suitable for specification of embedded

☆ This work has been partially supported by NSF grants CCR-9624324 and CCR-9615088.

* Corresponding author.

0167-6423/00/$ - see front matter
PII: S0167-6423(99)00018-0

software [17]. Embedded software is software that is part of a larger system and usually provides at least partial control over the system in which it is embedded. This type of software is often *reactive* in that it must react or respond to environmental conditions as reflected in the inputs arriving at the software boundary [17]. A *robust* system will detect and respond appropriately to violations of assumptions about the system environment (such as unexpected inputs). Therefore, in a robust system the software behavior must be completely specified with respect to its input domain. Furthermore, a system must be consistent; there should be no conflicting behaviors specified.

Several research groups have extensively studied analysis for these two properties. Heimdahl et al. refers to the properties as completeness (a response is specified for every possible input) and consistency (at the most, one response is specified for each input) [18]. Heitmeyer et al. refer to the same properties as *coverage* and *disjointness*. The problems involved when analyzing large specifications for these properties are the topic of this paper.

In state-based languages such as Statecharts, SCR, and RSML, the transitions between states are guarded by conditions; the guarding condition must be true before the transition can be taken. Completeness and consistency in a state-based model implies the following [18]:

(1) Every state must have a deterministic behavior (transition) defined for every possible input event.

(2) The logical OR of the guarding conditions on every transition out of any state must form a tautology; for any condition, there is always at least one transition that can be taken.

(3) The logical AND between the guarding conditions on two transitions out of a state must form a contradiction; for each possible condition, there is at most one feasible transition out of every state.

Thus, verifying consistency and completeness in state-based requirements primarily involves calculating the AND and OR of the guarding conditions on the transitions to see if they form contradictions and tautologies.

Manually verifying that a specification is complete and consistent is a time-consuming and error-prone process. Therefore, we formalized and implemented the analysis procedures in a prototype tool [18]. A detailed description of the tool is given in Section 4. To demonstrate that our automated approach was feasible and that it was relevant to realistic systems, we applied the analysis to the requirements for a large commercial avionics system called TCAS II (Traffic alert and Collision Avoidance System II) [24]. TCAS II is an airborne collision-avoidance system required on all commercial aircraft carrying more than 30 passengers through US airspace. TCAS II has been described by the head of the TCAS program at the FAA as one of the most complex systems to be incorporated into the avionics of commercial aircraft. Therefore, TCAS II provides a challenging testbed for experimental application of formal methods and automated analysis to a real system.

The initial results from the analysis effort were encouraging [18] and scaled well to a large requirements specification. Nevertheless, as our analysis efforts progressed some

limitations with our original approach surfaced. Most importantly, the accuracy of the algorithms we used to conjoin and disjoin large Boolean formulas needed improvement. In our first prototype tool, we abstracted all formulas to propositional logic and used binary decision diagrams (BDDs) [7] to manipulate the formulas. The approach provided excellent performance in terms of execution time and space requirements. Unfortunately, with this rather coarse abstraction technique the accuracy of the BDD approach was, in some cases, inadequate. When analyzing the most complex parts of the TCAS requirements, the number of spurious (false) error reports turned out to be a hindrance.

The problem with spurious error reports is not unique to our analysis approach. Other approaches to static analysis of software requirements, such as the approach used to analyze SCR style requirements for consistency [20], have similar problems. To overcome these problems, most approaches to static analysis of software specifications enforce restrictions on the modeling language to facilitate accurate analysis, such as restricting variables to Boolean [8, 9, 20].

In our work we want to avoid enforcing confining restrictions on our modeling language. RSML was successfully used to model a complex avionics system [23], and our experience from that effort convinced us that enforcing restrictions, such as restricting the variables to Boolean and Enumerated types, will limit the usability of the modeling language to a point where it will not be widely used by practitioners. Furthermore, since we intend this type of analysis to be used by practitioners in industrial projects, the analysis approach must be (1) automated, so that performing analysis of a requirements specification will not require elaborate manual abstractions and expensive training, (2) efficient, so that the analysis will be used on a regular basis, and (3) accurate, so that the real problems are not obscured by spurious error reports. Our initial approach using binary decision diagrams was fully automated and highly efficient, but for the most complex parts of our case study the abstraction to propositional logic led to large numbers of spurious error reports.

To overcome this problem we began investigating what classes of decision procedures we would need to incorporate to achieve adequate accuracy while at the same time maintain a high degree of automation. As we refined our tools we quickly realized that to reduce the number of spurious error reports to an acceptable level, we would have to extend our tool with decision procedures for at least Presburger arithmetic. We came to the conclusion that to complete our task we would eventually need the full power of a general purpose theorem prover. To avoid the huge investment in time required to build such an analysis tool we simply decided to adopt an existing tool for our analysis. We chose to work with PVS (Prototype Verification System) [11, 27, 28]. In this paper, we discuss the problems with spurious error reports and describe our experiences using the Prototype Verification System to reduce the number of abstractions and, thus, increase the accuracy of the analysis results.

The remainder of the paper is organized as follows. Sections 2 and 3 contain a short description of the RSML modeling language and an overview of our testbed – TCAS II. Section 4 describes our early tool, discusses the reasons behind our problems

with spurious error reports, and outlines alternative solutions to the problem. Section 5 discusses our experiences with using PVS for completeness and consistency analysis. Related work is discussed in Section 6, and Section 7 summarizes our experiences and provides some suggestions for future directions in static analysis of software specifications.

2. Requirements State Machine Language (RSML)

RSML was developed as a requirements specification language for embedded systems [23]. The language is based on hierarchical finite state machines and is similar to David Harel's Statecharts [14, 17]. For example, RSML supports parallelism, hierarchies, and guarded transitions.

One of the main design goals of RSML was readability and understandability by non-computer professionals such as users, engineers in the application domain, managers, and representatives from regulatory agencies. An RSML specification consists of a collection of *states*, *transitions*, *variables*, *interfaces*, *functions*, *macros*, and *constants* which will be discussed in the remainder of this section.

States are organized in a hierarchical fashion as in Statecharts. RSML includes three different types of states – *compound* states, *parallel* states, and *atomic* states. Atomic states are analogous to those in traditional finite state machines. Parallel states are used to represent the inherently parallel or concurrent parts of the system being modeled. Finally, compound states are used both to hide the detail of certain parts of the state machine so as to make the resulting model easier to comprehend, and to encapsulate certain behaviors in the machine.

For example, consider the simple system of a railroad crossing pictured in Fig. 1. The state hierarchy modeling this system could be represented as in Fig. 2. This representation includes all three types of states. *Train_Crossing* is a parallel state with five direct children; namely, *North_Lights*, *South_Lights*, *North_Gate*, *South_Gate*, and *Intersection*. All of these are compound states, most of which happen to contain only atomic states (*Up*, *Off*, etc.). Naturally, the state machine in a real system specification is never as simple as in Fig. 2. As an example of a realistic model, a part of the state machine modeling an intruding aircraft in TCAS II is shown in Fig. 8.

Transitions in RSML control the way in which the state machine can move from one state to another. The general form of any transition in RSML is shown in Fig. 3. As the figure shows, a transition consists of a source state, a destination state, a trigger event, a guarding condition, and a set of events that is produced when the transition is taken. In order to take an RSML transition, the following must be true: (1) the source state must be currently active, (2) the trigger event must occur while the source state is active, and (3) when the trigger event occurs, the guarding condition must evaluate to true. If all of these conditions are satisfied then the destination state will become active, the source state will become inactive, and the set of events **E** will be produced.

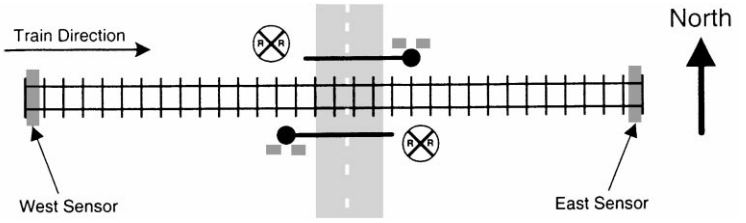

Fig. 1. Train crossing system.

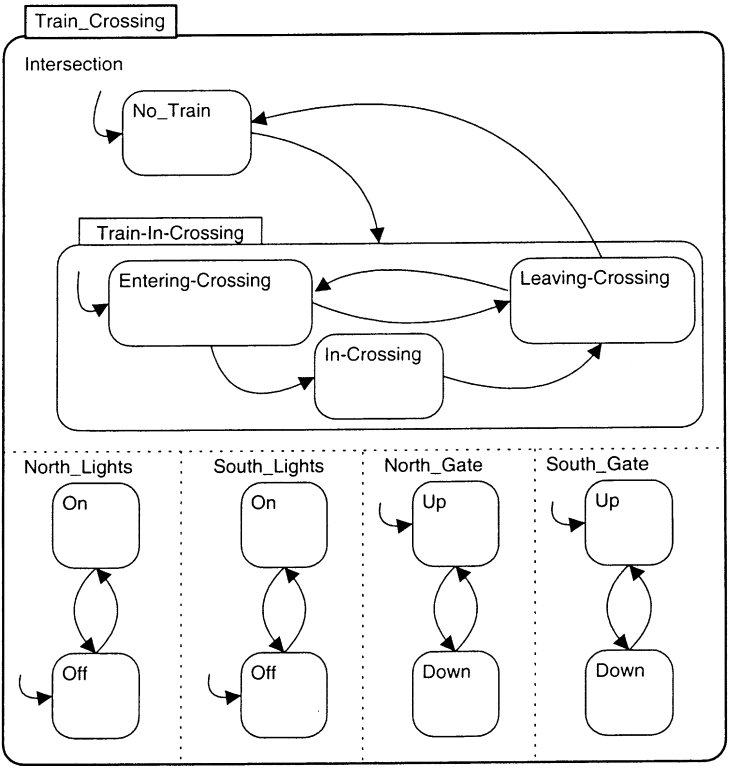

Fig. 2. Train crossing state machine.

The guarding condition is simply a predicate logic statement over the various states and variables in the specification; however, during the TCAS project, the team that developed the specification (the Irvine Safety Research Group led by Dr. Nancy Leveson) discovered that the guarding conditions required to accurately capture the requirements were often complex. The propositional logic notation traditionally used to define these conditions did not scale well to complex expressions and quickly became unreadable. To overcome this problem, they decided to use a tabular representation of disjunctive

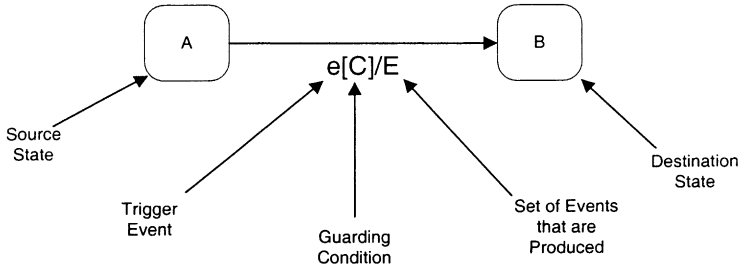

Fig. 3. General RSML transition.

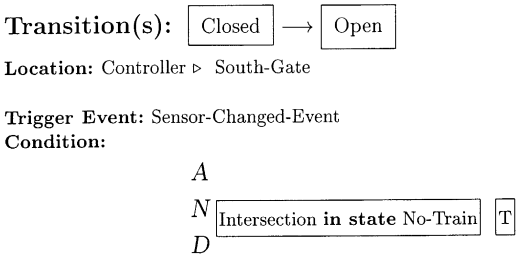

Fig. 4. A transition for the train crossing system.

normal form (DNF) that they called AND/OR tables (see Fig. 4 for a transition in the train crossing state machine and Fig. 5 for an example from the TCAS II requirements). 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. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated columns. A dot denotes "don't care".

Variables in the specification allow the analyst to record the values reported by various external sensors (in the case of input variables) and provide a place to capture the values of the outputs of the system prior to sending them out in a message (in the case of output variables).

To further increase the readability of the specification, the Irvine Group introduced many other syntactic conventions in RSML. For example, they allow expressions used in the predicates to be defined as functions (e.g., Other-Tracked-Alt$_{f\text{-}243}$), and familiar and frequently used conditions to be defined as macros (e.g., Threat-Condition$_{m\text{-}224}$). [1] *Functions* in RSML are mathematical functions that are used to abstract complex

[1] The subscript is used to indicate the type of an identifier (f for functions, m for macros, and v for variables) and gives the page in the TCAS II requirements document where the identifier is defined.

Transition(s): ESL-4 ⟶ ESL-2

Location: Own-Aircraft ▷ Effective-$SL_{s\text{-}30}$

Trigger Event: Auto-SL-Evaluated-$Event_{e\text{-}279}$

Condition:

				OR
	Auto-$SL_{s\text{-}30}$ **in state** ASL-2	T	T	·
A	Auto-$SL_{s\text{-}30}$ **in one of** {ASL-2,ASL-3,ASL-4,ASL-5,ASL-6,ASL-7}	·	·	T
N	Lowest-$Ground_{f\text{-}241}$ = 2	·	·	T
D	Mode-Selector = **one of** {TA/RA,TA-Only,3,4,5,6,7}	T	·	T
	Mode-$Selector_{v\text{-}34}$ = TA-Only	·	T	·

Output Action: Effective-SL-Evaluated-$Event_{e\text{-}279}$

Fig. 5. A transition definition from TCAS II with the guarding condition expressed as an AND/OR table.

calculations. A *macro* is simply a named AND/OR table that is used for frequently repeated conditions and is defined in a separate section of the document.

2.1. Formal semantics and analysis criteria

To facilitate automated analysis, the formal semantics of RSML has been defined as a composition of simple mathematical functions [18]. The behavior of a finite-state machine can be defined using a next-state relation. In RSML, this relation is modeled by the transitions between states and the sequencing of events.

In short, an RSML specification is a mapping from a set of states (called the set of all configurations – *Config*) representing the states in the graphical model and a set of variables (V) representing the input and output variables in the model, to new states and variables. Thus, the next state relation F is a mapping $\mathbb{C} \mapsto \mathbb{C}$, where $\mathbb{C} \subseteq (Config \times V)$. For a rigorous treatment of the formal foundation of RSML the interested reader is referred to [18]. A detailed description of the graphical notation and an account of the experiences from the TCAS II effort can be found in [23].

The individual transitions in the state machine are viewed as functions mapping one system state to the next. The structure of the state machine and the event propagation mechanism are then used to compose these functions into a statement about the behavior of the complete state machine. If a set of simple restrictions on the compositions are satisfied, it is guaranteed that the machine behaves as a function, and, by definition, is complete (a behavior has been defined for all possible inputs and input sequences) and consistent (no conflicting requirements exist). The interested reader is referred to [18] for a formal description of the compositional rules. In this paper it suffices to state the compositional rules informally as simple restrictions on the state machine:

(1) Every event generated as an action on a transition must trigger a transition somewhere else in the model; no events are lost or unaccounted for.

(2) There must be no information sharing between state machines executing in parallel; the execution of the machines is deterministic.

(3) Every pair of transitions out of the same state must have mutually exclusive guarding conditions; at most one transition can be taken at any time.

(4) The logical OR of the guarding conditions must form a tautology; if an event is generated there is at least one transition to take.

Analysis procedures assuring that the above criteria are satisfied are straightforward to automate: states are annotated with the events that trigger transitions out of the state so that all events can be accounted for, potential parallelism is detected through easily generated uses relations, a pairwise AND of the guarding conditions on transitions (triggered by the same event) out of a state must form a contradiction, and the disjunction of the guarding conditions on the transitions (triggered by the same event) out of a state must form a tautology. Clearly, the most costly part of the analysis is to AND and OR large guarding conditions together to check for contradictions and tautologies. The problems involved in manipulating these guarding conditions are the focus of this paper.

3. The case problem – TCAS II

TCAS is a family of airborne devices that function independently of the ground-based air traffic control (ATC) system to provide collision avoidance protection. TCAS II provides traffic advisories and recommended escape maneuvers (resolution advisories) in a vertical direction to avoid conflicting aircraft. In 1981, the FAA decided to develop and implement TCAS II, and a Minimal Operational Performance Standard (MOPS) document was produced that contained a mix of English description and psuedocode. This document was found to be lacking in several areas. It was not a requirements document but rather included much information that could be considered design or implementation detail. Due to the difficulties of FAA certification without a real system or system requirements, an effort started in 1990 to develop a requirements specification for the TCAS system. The formal RSML specification was developed in parallel with another, more traditional requirements effort that was later abandoned.

Since 1995, TCAS II has been included in all commercial aircraft carrying more than 30 passengers. TCAS II has been described by FAA officials as the most complex system ever to be incorporated into the avionics of commercial aircraft. The RSML specification of the TCAS system is the official requirements document and is used for FAA certification of manufacturers' TCAS implementations. Therefore, the TCAS specification provides a large, complex, and realistic testbed for our techniques.

In this paper we will use examples from the part of the collision avoidance system (CAS) in TCAS II that classifies intruding aircraft as Other-Traffic, Proximate-Traffic, Potential-Threats, or Threats.

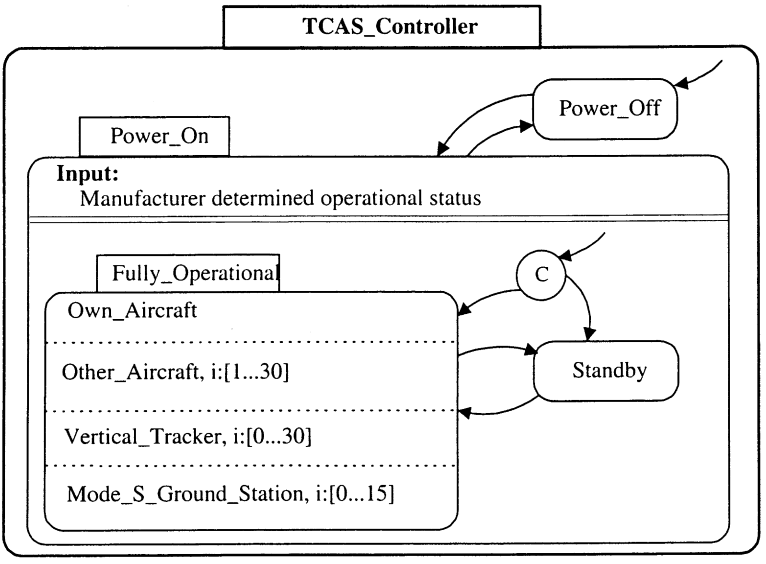

Fig. 6. Collision avoidance system highest level.

The highest level CAS state machine is shown in Fig. 6. At this level, CAS is either on or off; if it is on, it may be either fully operational or in standby mode. In the CAS logic, the states of five components are modeled: our own aircraft, other aircraft, mode-S ground radar stations, and a vertical tracker.

Fig. 7 shows the expanded Own-Aircraft portion of the CAS model. Effective-SL (sensitivity level) controls the dimensions of the protected airspace around own aircraft.

There are two primary means that CAS uses to determine Effective-SL: ground-based selection and pilot selection. When the pilot selects an automatic sensitivity selection mode, CAS selects sensitivity level based on the current altitude of own aircraft (defined in the Auto-SL state machine).

Alt-Layer effectively divides vertical airspace into layers (e.g., Layer-3 is approximately equal to the range 20 000–30 000 ft). Alt-Layer and Effective-SL are used in the determination of individual other aircraft threat classification (Fig. 8).

The state machine defining how an intruding aircraft is modeled in TCAS II is shown in Fig. 8. In short, the top-level state machine reflects whether a particular Other-Aircraft is currently being tracked or not.

The Intruder-Status state within Tracked reflects the current classification of Other-Aircraft (Other-Traffic, Proximate-Traffic, Potential-Threat, or Threat). When an intruder is classified as a threat, a two-step process is used to select a Resolution Advisory (RA). The first step is to select a sense (Climb or Descend). Based on the range and altitude tracks of the intruder, the CAS logic models the intruder's path until Closest Point of Approach (CPA). The CAS logic computes the predicted vertical separation for both climb and descend maneuvers, and selects the sense that provides the greater vertical separation.

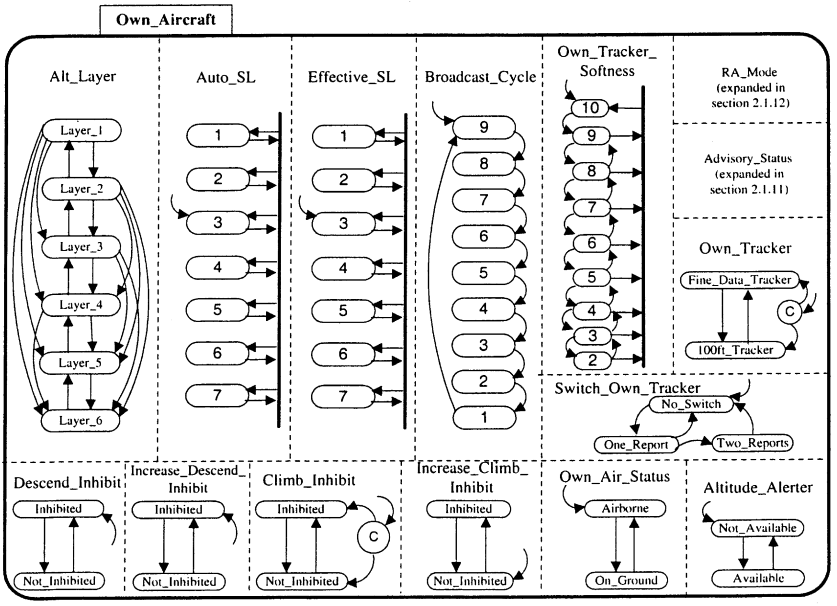

Fig. 7. Model of Own-Aircraft.

The second step in selecting an RA is to select the strength of the advisory. The least disruptive vertical rate maneuver that will still achieve safe separation is selected. For a more complete description of TCAS II and how it was modeled using RSML the reader is referred to [23].

The Vertical-Tracker models the details of the tracking algorithms used to track Other-Aircraft. The Mode-S-Ground Station models ground radar stations. Although theoretically the CAS logic uses input from the ground stations, these are not operational at this time. The Vertical-Tracker and Mode-S-Ground Station models are not needed to understand the basic function of TCAS II and are not explained further here.

4. Analyzing for completeness and consistency

As mentioned in the previous section, analyzing state-based specifications for completeness and consistency involves conjoining guarding conditions together to see if they are contradictory, and disjoining conditions to see if they form tautologies.

During our work analyzing TCAS II we developed several generations of analysis tools for completeness and consistency analysis. In this section we provide an overview of the evolution of our tools, point out where the various approaches have fallen short, and discuss some observations on the analytical power needed in a successful static analysis tool for the analysis of software specifications.

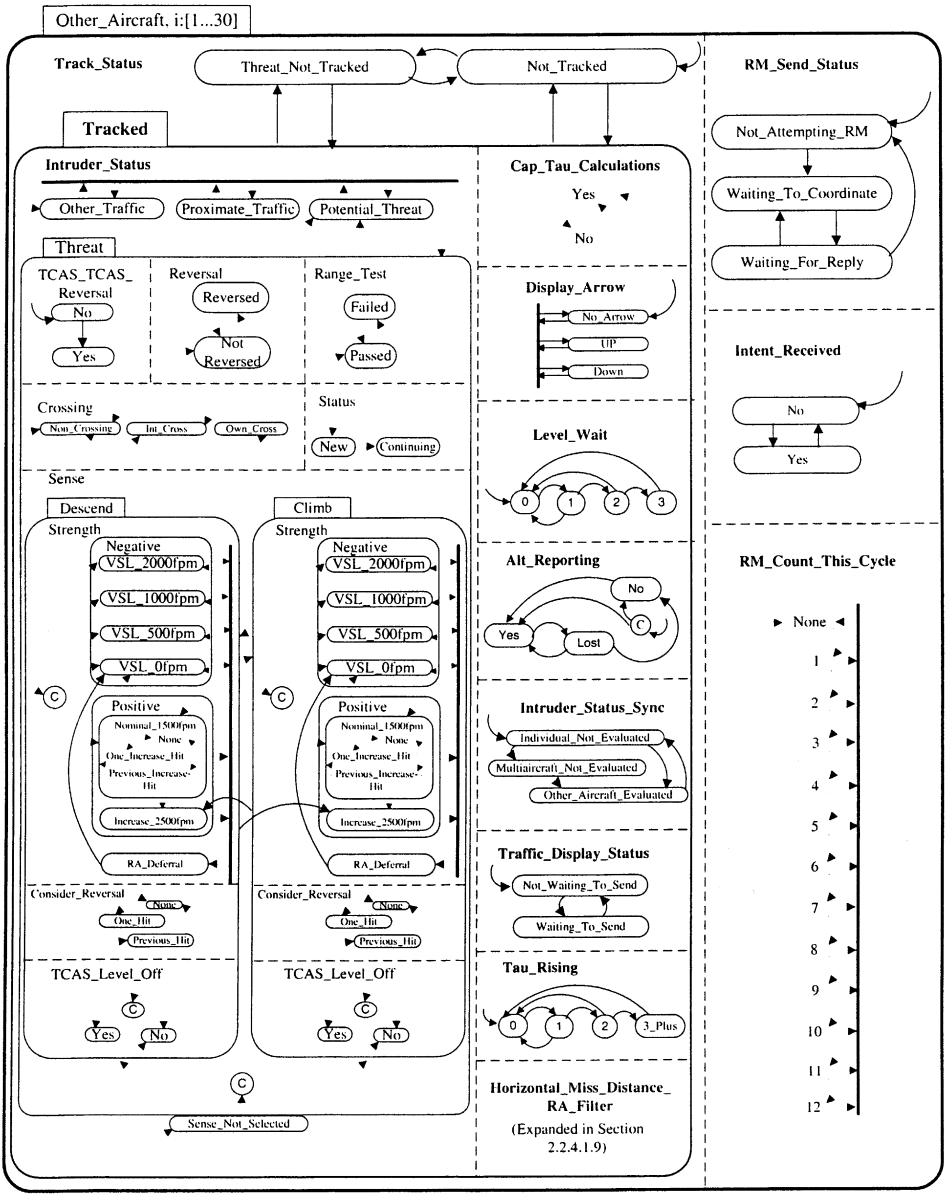

Fig. 8. The model of the behavior of an intruding aircraft.

4.1. The "Early" approach

Our first prototype tool parsed a textual version of RSML, built an internal representation of the state machine appropriate for execution as well as analysis, and performed completeness and consistency analysis [18]. The tool also generated other information,

Transition(s): Proximate-Traffic ⟶ Other-Traffic

Location: Other-Aircraft ▷ Intruder-Status$_{s\text{-}136}$

Trigger Event: Air-Status-Evaluated-Event$_{e\text{-}279}$

Condition:

		OR							
	Alt-Reporting$_{s\text{-}101}$ **in state** Lost	T	T	·	·	·	·	·	·
	Alt-Reporting$_{s\text{-}101}$ **in state** No	·	·	T	T	T	T	·	·
	RA-Mode-Cancelled	·	·	T	T	·	·	·	·
A	Other-Bearing-Valid$_{v\text{-}120}$ = True	F	·	F	·	F	·	·	·
N	Other-Range-Valid$_{v\text{-}117}$ = True	·	F	·	F	·	F	·	·
D	Proximate-Traffic-Condition$_{m\text{-}216}$	·	·	·	·	·	·	F	·
	Potential-Threat-Range-Test$_{m\text{-}15}$	·	·	·	·	T	T	·	·
	Potential-Threat-Condition$_{m\text{-}213}$	·	·	·	·	·	·	F	·
	Threat-Condition$_{m\text{-}224}$	·	·	·	·	·	·	F	·
	Other-Air-Status$_{s\text{-}101}$ **in state** On-Ground	·	·	·	·	·	·	·	T

Output Action: Intruder-Status-Evaluated-Event$_{e\text{-}279}$

Fig. 9. The transition downgrading an intruder from being in the proximity to being considered other traffic.

such as uses hierarchies, event propagation information, and circular dependencies in macros and functions, and allowed execution and animation of the state machines.

Initially, our main concern was the performance of the AND and OR operations needed to check for mutual exclusion and complete coverage. The tables representing the guarding conditions are often large (10–15 rows; Fig. 9) and reference macros through several levels of indirection. By indirection we mean that one table references one or more macros that in their turn reference other macros. Three to six levels of indirection in the guarding conditions are common. An example of indirection is shown in Fig. 9, where the guarding condition refers to the macro Potential-Threat-Range-Test shown in Fig. 10. Consequently, a large table typically uses 50–70 predicates spread over 10–15 macros.

As previously mentioned, in our initial tool, we abstracted all formulas to propositional logic and used binary decision diagrams (BDDs) [7] to represent and manipulate the guarding conditions. BDDs are directed acyclic graphs that represent Boolean functions in a canonical form. Algorithms for manipulating BDDs, for example, conjoining and disjoining formulas, are efficient and provide good average performance [7].

We used structural equivalence to determine if two predicates were the same. Thus, we did not interpret any of the relationships between the predicates in the model. Given this encoding, the tables are then translated to BDDs and all analysis is performed in the BDD domain. If an error is detected, for instance, an AND operation (checking if the guarding conditions on two transitions are mutually exclusive) does not reduce to the

Macro: Potential-Threat-Range-Test
Definition:

		OR			
	$\text{Other-Tracked-Range}_{\text{f-245}} > 0$	·	T	T	F
	$\text{Other-Tracked-Range-Rate}_{\text{f-245}} > 10\ \text{ft/s}_{(\text{RDTHRTA})}$	T	F	F	F
	$\text{Other-Tracked-Range}_{\text{f-245}} * \text{Other-Tracked-Range-Rate}_{\text{f-245}} \leq \text{H1TA}$	T	·	·	·
A	$\text{Other-Tracked-Range}_{\text{f-245}} \leq \text{DMODTA}$	T	·	·	·
N	$\text{Other-Tracked-Range-Rate}_{\text{f-245}} < -10\ \text{ft/s}_{(\text{RDTHRTA})}$	·	T	F	·
D	$\frac{\frac{\text{DMODTA}^2}{\text{Other-Tracked-Range}_{\text{f-245}}} - \text{Other-Tracked-Range}_{\text{f-245}}}{\text{Other-Tracked-Range-Rate}_{\text{f-245}}} < \text{TRTHRTA}$	·	T	·	·
	$\frac{\frac{\text{DMODTA}^2}{\text{Other-Tracked-Range}_{\text{f-245}}} - \text{Other-Tracked-Range}_{\text{f-245}}}{-10\ \text{ft/s}_{(\text{RDTHRTA})}} < \text{TRTHRTA}$	·	·	T	·
	$0\text{s}_{(\text{MINTAU})} < \text{TRTHRTA}$	·	·	·	T

Fig. 10. The macro defining the potential threat range test.

constant FALSE, the analysis result is translated back to an AND/OR table and presented to the analyst as an error report.

For example, consider Fig. 11. In TCAS, the concept of sensitivity level is used to determine how close an intruder is allowed to get before an advisory is presented to the pilot. A higher sensitivity level indicates a more sensitive setting of TCAS II; a more sensitive setting means an advisory will be generated earlier (while the planes are farther apart). The bar on the side of the states in the figure is a transition bus. Many state machines in the model were found to be fully interconnected, that is, there were transitions between all the states in the machine; the transition bus was introduced to make the graphical representation cleaner.

An inconsistency can be detected between the transitions ESL-4 → ESL-2 (Fig. 5) and ESL-4→ESL-5 (Fig. 12). The inconsistency (as reported from the initial analysis tool) is shown in Fig. 13: Column 3 in Fig. 5 and column 3 in Fig. 12 are both satisfied by the condition in the error report. Since sensitivity level ESL-5 represents a sensitive setting and ESL-2 represents that advisories are turned off (no warnings are given to the pilot), a potentially hazardous inconsistency is present. After an evaluation of the inconsistency, it was determined that the guarding condition on the transition to ESL-2 was too weak and needed strengthening. The strengthened condition can be seen in Fig. 14. With this strengthening of the guarding condition on the transition to ESL-2 the transitions are consistent.

The BDD approach has provided excellent performance (in terms of execution time and memory requirements) for all examples from TCAS we analyzed in this initial

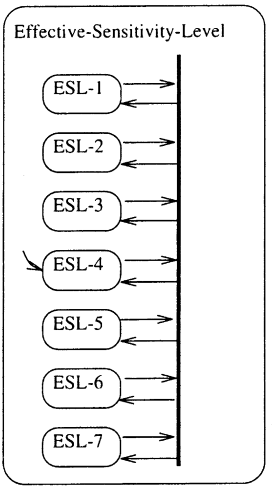

Fig. 11. Effective sensitivity level.

Transition(s): ESL-4 ⟶ ESL-5

Location: Own-Aircraft ▷ Effective-$SL_{s\text{-}30}$

Trigger Event: Auto-SL-Evaluated-$Event_{e\text{-}279}$
Condition:

						OR	
	Auto-$SL_{s\text{-}30}$ **in state** ASL-5	T	T	T	·	·
	Auto-$SL_{s\text{-}30}$ **in one of** {ASL-5,ASL-6,ASL-7}	·	·	·	T	T
	Lowest-$Ground_{f\text{-}241}$ = **one of** {5,6,7,None}	T	·	·	·	T
A	Lowest-$Ground_{f\text{-}241}$ = 2	·	·	T	·	·
N	Lowest-$Ground_{f\text{-}241}$ = 5	·	·	·	T	·
D	Mode-Selector = **one of** {TA/RA,5,6,7}	T	·	·	T	·
	Mode-$Selector_{v\text{-}34}$ = TA-Only	·	T	·	·	·
	Mode-Selector = **one of** {TA/RA,TA-Only,3,4,5,6,7}	·	·	T	·	·
	Mode-$Selector_{v\text{-}34}$ = 5	·	·	·	·	T

Output Action: Effective-SL-Evaluated-$Event_{e\text{-}279}$

Fig. 12. The transition from Effective-SL state ESL-4 to ESL-5.

case study [18]. However, since we abstracted all conditions to propositional logic and, thus, failed to take their possible interrelationships into account; the analysis may report spurious errors. A spurious error report is a report that, for example, two transitions are inconsistent when the conditions a and b are both true; a and b, however, may

```
ESL_4 --> ESL_2 conflicts with ESL_4 --> ESL_5 if

Auto_SL In State ASL_2                              : F
Auto_SL In One Of {ASL_2,ASL_4,ASL_5,ASL_6,ASL_7}   : T
Lowest_Ground() == 2                                : T
Mode_Selector Eq. One Of {TA_RA,TA_Only,3,4,5,6,7}  : T
Auto_SL In State ASL_5                              : T
```

Fig. 13. Consistency analysis results for Effective-SL state ESL-4.

Transition(s): ESL-4 ⟶ ESL-2

Location: Own-Aircraft ▷ Effective-$SL_{s\text{-}30}$

Trigger Event: Auto-SL-Evaluated-$Event_{e\text{-}279}$
Condition:

A N D	Auto-$SL_{s\text{-}30}$ **in state** ASL-2	T
	Mode-$Selector_{v\text{-}34}$ = Standby	F

Output Action: Effective-SL-Evaluated-$Event_{e\text{-}279}$

Fig. 14. The modified transition from Effective-SL ESL-4 to ESL-2.

be mutually exclusive so the inconsistency is never satisfiable and should not have been reported. But, since the functions in a and b are abstracted to propositions, the contradiction is not detected and the condition is reported as an error.

4.2. The problem with spurious errors

During additional case studies with our first prototype tool, spurious error reports were not a serious problem [18]. All spurious errors could be traced either to (1) a lack of a type system in RSML or 2 the inability of the tool to adequately include information about the structure of the state machine in the analysis. For example, consider the conditions

Auto-$SL_{s\text{-}30}$ **in state** ASL-5 and Auto-$SL_{s\text{-}30}$ **in state** ASL-2

(appearing in Figs. 5 and 12). Without the information that Auto-Sensitivity-Level can only be in one state at a time, the tool would generate spurious error reports and indicate that additional conflicts between the two transitions existed when they, in fact, did not. For example, the error report below would be generated by the tool (indicating a conflict exists between column 1 in Fig. 5 and column 2 in Fig. 12).

```
ESL_4 --> ESL_2 conflicts with ESL_4 --> ESL_5 if

  Auto_SL In State ASL_2   : T
  Auto_SL In State ASL_5   : T
  Mode_Selector == TA_Only : T
```

This is clearly an erroneous report because the state machine Auto-Sensitivity-Level can be in at most one state at a time; we do not need to consider any conflicts under the unsatisfiable condition shown above. Similar problems relating to enumerated data types also led to spurious error reports. These drawbacks were trivial to address by augmenting the tool with decision procedures to handle the type system and formulas related to the hierarchical structure of the state machine. The decision procedures were invoked in the translation from a BDD to an AND/OR table. An updated version of the tool eliminated all these problems with spurious errors.

Unfortunately, decision procedures for enumerated types and the state hierarchy do not eliminate all spurious error reports we encountered in our work. To make RSML usable on large real-life projects, we allow input and output variables of types Boolean, enumerated, integer, and real, and we allow arbitrary arithmetic expressions to appear in the predicates used in the AND/OR tables (see, for example, rows 6 and 7 in Fig. 10). These features complicate the analysis and in some cases make the analysis undecidable. These complications could be eliminated by limiting the expressive power of the language, for example, by prohibiting the use of quantification, multiplication, and division, but this would, in our opinion, make the language too restricted to be useful for the modeling of realistic software applications. Instead, we are attempting to find effective, accurate, and automated analysis procedures, while at the same time maintaining a language suitable for the specification of complex real world software systems.

4.3. Improving the tool

Clearly, predicates involving, for example, inequalities and arithmetic expressions, cannot be handled with the simple decision procedures discussed thus far. For instance, consider the state machine Descend-Inhibit in Fig. 15. The guarding conditions on the two transitions out of the state Inhibited (Fig. 16) contain simple arithmetic expressions, and our simple decision procedures will fail to detect that the transitions out of Inhibited are both complete and consistent. Instead, our tool will generate the spurious error reports shown in Fig. 17. For simple transition predicates (such as the ones in the example) this is not a problem. An analyst can simply eliminate the spurious error reports manually. However, when analyzing the most complex parts of TCAS II, the number of spurious error reports can be large, and manual inspection of the results becomes infeasible.

Consider the transition in Fig. 9 and a condition of similar size from another transition (not shown in this paper). When analyzing transitions from this part of TCAS II the BDDs representing the analysis results typically contained more than 1000 nodes

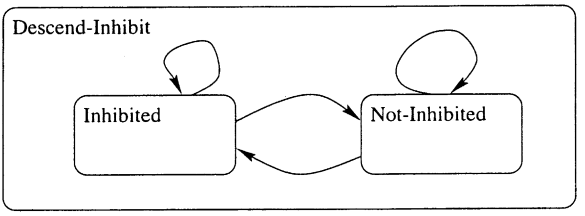

Fig. 15. The state machine for Descend-Inhibit.

Transition(s): Inhibited ⟶ Not-Inhibited

Location: Own-Aircraft ▷ Descend-Inhibit$_{s\text{-}30}$

Trigger Event: Surveillance-Complete-Event$_{e\text{-}279}$
Condition:

(Own-Tracked-Alt$_{f\text{-}248}$ - Ground-Level$_{f\text{-}237}$) > 1200ft$_{(NODESHI)}$ T

Output Action: Descend-Inhibit-Evaluated-Event$_{e\text{-}279}$

Transition(s): Inhibited ⟶ Inhibited

Location: Own-Aircraft ▷ Descend-Inhibit$_{s\text{-}30}$

Trigger Event: Surveillance-Complete-Event$_{e\text{-}279}$
Condition:

Own-Tracked-Alt$_{f\text{-}248}$ ≤ (1200ft$_{(NODESHI)}$ + Ground-Level$_{f\text{-}237}$) T

Output Action: Descend-Inhibit-Evaluated-Event$_{e\text{-}279}$

Fig. 16. The transitions out of Inhibited.

and the resulting error report presented as an AND/OR table contained thousands (in many cases hundreds of thousands) of columns. Error reports of this size are clearly too large for any form of manual inspection and improvements in analysis accuracy are needed.

To overcome this problem we began implementing decision procedures for equivalence and inequivalence for expressions with structurally equivalent left- and right-hand sides. Unfortunately, as shown in Fig. 16, many expressions may not be structurally equivalent. In a large specification, such as the TCAS II specification, an expression may be captured as $a + b$ in one part of the document and as $b + a$ in another part.

```
Inhibited --> Inhibited conflicts with
   Inhibited --> Not-Inhibited if

  (Own_Tracked_Alt() - Ground_Level()) > 1200  : T ;
  Own_Tracked_Alt() <= (1200 + Ground_Level()) : T ;


No transition out of Inhibited is satisfied if :

  (Own_Tracked_Alt() - Ground_Level()) > 1200  : F ;
  Own_Tracked_Alt() <= (1200 + Ground_Level()) : F ;
```

Fig. 17. Spurious error reports generated for the state Inhibited.

Attempting to enforce consistency in a team of tens of analysts working on an evolving document several hundred pages long is extremely difficult at best. Thus, relying on structurally equivalent expressions is unrealistic.

Naturally, limiting the input and output variables to enumerated types and disallowing arithmetic expressions in the predicates would eliminate all of our problems with spurious errors. Unfortunately, it would also make RSML unsuitable as a modeling language for embedded control systems, the domain for which the language was originally developed [23]. In this domain we often want to use simple arithmetic expressions when defining the guarding conditions, for instance, we may want to change state if the average of three sensor readings is below a setpoint or if the difference between two consecutive reports from a sensor is within a certain hysteresis factor. In short, since we do not find it feasible to limit the expressive power of the language we must attempt to find more powerful analysis methods.

At this point, it became clear that to provide the analytical power needed to work with TCAS II we would have to extend our tool with decision procedures for at least Presburger arithmetic. To avoid the huge investment in time required to build such a tool, we simply decided to adopt a powerful existing tool for our analysis. We chose to work with the theorem prover Prototype Verification System (PVS) [11, 27, 28] and simply output proof obligations for our completeness and consistency criteria.

Naturally, there are other approaches to addressing the problems with arithmetic expressions. One way is to use our BDD tool to produce simple arithmetic verification conditions and use decision procedures for Presburger arithmetic that can be decided without a general purpose theorem prover [31, 38, 40]. This is the approach we initially pursued.

A second approach is to refine our abstraction mechanism. In this case, the abstraction function is applied before the translation to a BDD. For example, the arithmetic expressions may be used to divide their domain to a finite number of subsets and the abstraction function can be defined to generate no spurious errors. This approach has been successfully used in model checking [10, 41].

There were two main reasons why we chose a theorem prover such as PVS instead of using some simpler tools or implementing our own solution. First, as we will show in the next section, some parts of the TCAS II analysis require the ability to reason about universal and existential quantification. Second, the various tools developed in the verification community are often targeted towards a specific problem and seemed to be quite difficult to integrate to a tool suited for the verification of the properties in which we were interested with respect to TCAS II.

5. Using PVS for completeness and consistency analysis

The PVS is a verification system that provides an interactive environment for the development and analysis of formal specifications. PVS consists of a specification language, a parser, a type-checker, an interactive theorem prover, and various browsing tools. PVS has been used to model and reason about many different systems, for example, clock synchronization [34, 36] and the AAMP5 Microprocessor [26].

PVS contains a large library (called a prelude file) that contains many built-in theories that can be used during the proof process. Some of the built-in theories included in the prelude file are Boolean properties, quantifier properties, equality properties, functions, sets, reals and real properties, and rationals and rational properties. In addition, PVS contains decision procedures for equality and linear inequality that are complete for linear arithmetic (multiplication by literal constants) on the reals [32, 37]. Finally, PVS is freely available and the tool is well supported. These factors made PVS an ideal candidate to solve our problems with the spurious errors outlined in the previous section.

5.1. Incorporating PVS

To incorporate PVS into our analysis environment, we augmented our existing tool with the capability to generate theories and proof obligations in the PVS specification language.

Our tool generates a PVS theory for each transition in an RSML specification. We do this in a two stage process. First, we define each predicate in the AND/OR table as a predicate in the PVS specification language.[2] Second, a predicate representing the full guarding condition is built from the individual predicates defined in the first stage.

For example, consider the transition from the state Inhibited to the state Not-Inhibited defined in Fig. 16. In PVS, this transition would be defined by the theory shown in Fig. 18. The constants in the system are defined as a separate theory and imported to the theory defining the transition. The functions Own-Tracked-Alt and Ground-Level used in the RSML specification are mapped to variables of type real in PVS. RSML

[2] A predicate in PVS is a function with return type Boolean.

```
%----------------------------------------%
%  Theory for transition from state      %
%  Inhibited to state Not_Inhibited      %
%----------------------------------------%
Inhibited_To_Not_Inhibited: THEORY
BEGIN

%----------------------------------------%
%  Import declarations for constants     %
%----------------------------------------%
IMPORTING Constants

%--------------------------%
%  Variable Declarations:  %
%--------------------------%
Own_Tracked_Alt, Ground_Level: VAR real

%--------------------------------------------------%
% Definition for the first (and in this case, only) %
% predicate in the guarding condition on the        %
% transition from state Inhibited to state          %
% Not_Inhibited.                                    %
%                                                   %
% NODESHI - Threshold for descend inhibit (const)   %
%--------------------------------------------------%
Pred168Grtr?(Own_Tracked_Alt, Ground_Level): bool=
   Own_Tracked_Alt - Ground_Level > NODESHI

%--------------------------------------------------%
% Interface to the theory for the transition from  %
% state Inhibited to state Not_Inhibited.          %
%--------------------------------------------------%
Trans_Inhibited_To_Not_Inhibited?(Own_Tracked_Alt, Ground_Level): bool=
   Pred168Grtr?(Own_Tracked_Alt, Ground_Level)

END Inhibited_To_Not_Inhibited
```

Fig. 18. A PVS theory for the transition from Inhibited to Not-Inhibited.

does not make any assumptions about the range of values a function can return, so all functions can be mapped to variables as long as the type is maintained.

The actual proof obligations are generated based on the criteria described in Section 2. In this evaluation, we use PVS to perform the AND and OR operations required for our analysis. To illustrate our approach, consider the two simple transitions out of the state Inhibited in Fig. 16. Given these transitions, the tool will generate the two proof obligations shown in Fig. 19. Naturally, the full power of PVS is not needed in these cases. A simpler tool based on decision procedures for Presburger arithmetic would have been ideal. In fact, after this investigation was completed David Park investigated how a tool called the Stanford Validity Checker (SVC) [4] could be used to analyze these parts of TCAS II [30].

```
Inhibited: THEORY
BEGIN
%-------------------------------------------------%
% Import declarations for theories for transitions %
%  - Inhibited_To_Not_Inhibited,                  %
%  - Inhibited_To_Inhibited,                      %
%-------------------------------------------------%
IMPORTING Inhibited_To_Not_Inhibited, Inhibited_To_Inhibited

%-------------------------%
%  Variable Declarations:  %
%-------------------------%
Own_Tracked_Alt, Ground_Level: VAR real

%-------------------------------------------------%
% Conjecture: The conditions for transitions out of %
% state Inhibited are completely specified.       %
% Cond1 \/ Cond2 -> True                          %
%-------------------------------------------------%

InhibitedToNotInhibited_InhibitedToInhibited_Complete: CONJECTURE
 Trans_Inhibited_To_Not_Inhibited?(Own_Tracked_Alt, Ground_Level)
OR
 Trans_Inhibited_To_Inhibited?(Own_Tracked_Alt, Ground_Level)

%-------------------------------------------------%
% Conjecture: The conditions for transitions out of %
% state Inhibited are consistent; non-conflicting.  %
% Cond1 /\ Cond2 -> False                         %
% rewritten as (not(Cond1 /\ Cond2)               %
%-------------------------------------------------%

InhibitedToNotInhibited_InhibitedToInhibitedConsistent: CONJECTURE
NOT
(Trans_Inhibited_To_Not_Inhibited?(Own_Tracked_Alt, Ground_Level
 &
 Trans_Inhibited_To_Inhibited?(Own_Tracked_Alt, Ground_Level))

END Inhibited
```

Fig. 19. Proof obligations generated for the state Inhibited.

5.2. *Quantification*

Statecharts as well as RSML allows identical state machines to be parameterized into *state arrays*. For instance, the Other-Aircraft state machine in Fig. 8 is a state array. In TCAS II we are required to be able to track up to 30 intruders. In effect, the Other-Aircraft state machine is 30 identical state machines existing in parallel. The array notation is simply a syntactic convenience allowing us to produce more compact models. Given the array construct it becomes necessary to use quantification over the state array when expressing guarding conditions. For example, we may want to take some action if *one or more* of the intruding aircraft are in the sate Threat or take some other action if *all* intruders are in the state Other-Traffic.

To illustrate the need for quantification and the full power of PVS we use an example from the specification of how TCAS II communicates with the pilot's display. The example involves the interface definition between the TCAS II box and the communications medium connected to the display. In this case the quantification is over a finite interval. In some situations, however, it may be necessary to quantify over an unbounded domain, for example, when modeling interaction with an unbounded number of input devices.

Interface definitions in RSML consist of two parts: (1) a physical interface definition that captures properties related to the physical aspects of the communication, for example, the channel name and simple timing assumptions, and (2) a collection of handlers that determine under which conditions we can send/receive messages over this channel. The physical interface definition is used to assure that components connected together have compatible properties, for example, that the expected arrival rate at the RECEIVE side is greater than or equal to the expected send rate at the SEND side.

The output interface in Fig. 20 is interpreted as follows. When a state machine in the RSML specification generates the interface's trigger event and one of the handler guarding conditions is satisfied, the output action in that handler is performed. In this example taken from TCAS II, the state machine model was required to model 30 intruding aircraft (modeled with state machines named Other-Aircraft). The model of each Other-Aircraft contains a state machine called Traffic-Display-Status. When TCAS has detected an intruder and has determined that the pilot needs to be notified, the state machine Traffic-Display-Status associated with that intruder will enter the state Waiting-To-Send. This indicates that TCAS is ready to send an advisory regarding this particular intruder to the pilot's display.[3] If TCAS tracks several intruders and needs to notify the pilot about more than one intruder (more than one Other-Aircraft model is in state Waiting-To-Send), the intruder model with the highest priority (Traffic-Score) takes precedence. The advisory relating to an intruder is contained in the variable Advisory-Code. The communication handler in Fig. 20 is parameterized. Any Other-Aircraft model can generate a trigger event for this handler. The handler will simply be instantiated with the index of that intruder (the index is indicated with the *i* in the definition). Thus, the interface in Fig. 20 tells us that Other-Aircraft[i] can only send an advisory to the pilot if there are no Other-Aircraft models ready to send (columns 1 and 2) or there are no Other-Aircraft models with a higher traffic score (columns 3 and 4). In addition, if we are to send an advisory and we are in state Threat, the advisory must be a Resolution-Advisory (rows 5 and 6). The second handler in Fig. 20 defines the case when TCAS will not send any advisory to the pilot's display.

The notion of completeness and consistency discussed earlier in this paper extends to the RSML interface definitions.

[3] An advisory is a notification to the pilot. For example, if the intruder is very close, a resolution advisory will be displayed. A resolution advisory is an advisory telling the pilot what action to take to avoid the potential collision, for example *Climb*! or *Descend*!

Output Interface: Display-Unit-Interface
Channel: Display-Channel
Trigger: Send-Traffic-Event[i]
Max Separation: 1.2 second
Min Separation: 0.8 second

Handler-1
Condition:
For all j **in** {1..30}:

		OR			
	i ≠ j	T	T	·	·
A	Traffic-Display-Status[i] **in state** Waiting-To-Send	T	T	T	T
N	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	F	·	·
D	Traffic-Score(Other-Aircraft[i]) ≥ Traffic-Score(Other-Aircraft[j])	·	·	T	T
	Other-Aircraft[i] **in state** Threat	F	T	F	T
	Advisory-Code[i] = Resolution-Advisory	·	T	·	T

Action: SEND(Advisory-Code[i])

Handler-2
Condition:
Exists at least one j **in** {1..30}:

		OR					
	i ≠ j	T	T	F	F	·	·
A	Traffic-Display-Status[i] **in state** Waiting-To-Send	F	F	F	F	T	T
N	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	F	·	·	T	T
D	Traffic-Score(Other-Aircraft[i]) ≥ Traffic-Score(Other-Aircraft[j])	·	·	·	·	F	F
	Other-Aircraft[i] **in state** Threat	F	T	·	F	F	·
	Advisory-Code[i] = Resolution-Advisory	·	T	T	·	·	T

Action: None

Fig. 20. Definition of the communication with the TCAS display.

(1) Within an interface definition, every pair of handlers must have mutually exclusive guarding conditions; at most one handler can be used at any time.

(2) The logical OR of the guarding conditions on all handlers within an interface definition must form a tautology; if an action is requested on a channel, it is always defined how this action will be handled.

The verification of completeness and consistency in the interfaces is identical to the verification for the rest of the state machines. In this situation, however, we are required to include the quantifiers in the PVS theories generated by our tool. The PVS theory for Handler-1 can be seen in Fig. 21. The proof obligation for consistency is shown in Fig. 22. Note that the indices for the arrays (i and j) have been included in this theory. Thus, to analyze certain parts of the TCAS II requirements, it was necessary and convenient to use universal and existential quantification.

```
%-----------------------------------------------%
%  Theory for Output Interface Handler 1 Condition %
%-----------------------------------------------%
Handler1_OutputInterface: THEORY
BEGIN
  IMPORTING TypeDefs

  AdvisoryCode: VAR AdvisoryCodeType
  TrafficDisplayStatus: VAR TrafficDisplayStatusType
  OtherAircraft: VAR OtherAircraftType
  TrafficScore: VAR TrafficScoreType
  j: VAR int
  i: VAR i_type
%--------------------------%
%  Predicate Declarations:  %
%--------------------------%
  pred1?(i, j): bool = NOT (i = j)
  pred2?(TrafficDisplayStatus, i): bool = WaitingToSend?(TrafficDisplayStatus(i))
  pred3?(TrafficDisplayStatus, j): bool = WaitingToSend?(TrafficDisplayStatus(j))
  pred4?(TrafficScore, OtherAircraft, i, j): bool =
         TrafficScore(OtherAircraft(i)) >= TrafficScore(OtherAircraft(j))
  pred5?(OtherAircraft, i): bool = Threat?(OtherAircraft(i))
  pred6?(AdvisoryCode, i): bool = ResolutionAdvisory?(AdvisoryCode(i))
%-----------------------------------------------------%
%  Handler 1 output interface condition specification.  %
%-----------------------------------------------------%
  Handler1?(AdvisoryCode, TrafficDisplayStatus, OtherAircraft, TrafficScore, i): bool =
    FORALL (j: int | j >= 1 AND j <= 30):
      (  (pred1?(i, j)
            & pred2?(TrafficDisplayStatus, i)
            & NOT pred3?(TrafficDisplayStatus, j)
            & NOT pred5?(OtherAircraft, i))
        OR
         (pred1?(i, j)
            & pred2?(TrafficDisplayStatus, i)
            & NOT pred3?(TrafficDisplayStatus, j)
            & pred5?(OtherAircraft, i)
            & pred6?(AdvisoryCode, i))
        OR
         (pred2?(TrafficDisplayStatus, i)
            & pred4?(TrafficScore, OtherAircraft, i, j)
            & NOT pred5?(OtherAircraft, i))
        OR
         (pred2?(TrafficDisplayStatus, i)
            & pred4?(TrafficScore, OtherAircraft, i, j)
            & pred5?(OtherAircraft, i)
            & pred6?(AdvisoryCode, i)))

  END Handler1_OutputInterface
```

Fig. 21. The PVS theory describing Handler-1 in Fig. 20.

5.3. Summary observations

For PVS to be considered as an alternative to our previous approach it must reduce the problem with spurious error reports and at the same time maintain acceptable performance in terms of execution time.

```
%----------------------------------------%
%  Conjecture Theory for Output Interface %
%----------------------------------------%
OutputInterface: THEORY
  BEGIN
%-------------------------------------------------------------%
%  Import the abstract data type definitions and the theories %
%  for the output interface handlers.                         %
%-------------------------------------------------------------%
     IMPORTING TypeDefs, Handler1_OutputInterface, Handler2_OutputInterface,
               Handler3_OutputInterface, Output_Invariant
%-------------------------%
%  Variable Declarations:  %
%-------------------------%
     AdvisoryCode: VAR AdvisoryCodeType
     TrafficDisplayStatus: VAR TrafficDisplayStatusType
     OtherAircraft: VAR OtherAircraftType
     TrafficScore: VAR TrafficScoreType
     i: VAR i_type
%-------------------------------------------------------------%
%  Conjecture to check handler 1 and handler 2 output interfaces %
%  for consistency.  The conjunction of the interface condition  %
%  for handler 1 and the interface condition for handler 2 must  %
%  be a contradiction.                                           %
%-------------------------------------------------------------%
     Handler1_and_Handler2_OutputInterface_Consistent: CONJECTURE
       NOT(Handler1?(AdvisoryCode, TrafficDisplayStatus, OtherAircraft, TrafficScore, i)
           &
           Handler2?(AdvisoryCode, TrafficDisplayStatus, OtherAircraft, TrafficScore, i))

  END OutputInterface
```

Fig. 22. The PVS theory describing the consistency proof obligation for the interface in Fig. 20.

The guarding conditions in the TCAS II specification generally fall into two categories: (1) relatively simple tables with 1–10 predicates and few predicates involving mathematical expressions, such as the tables in Figs. 5 and 12, and (2) very complex tables with 20 or more predicates and many predicates involving arithmetic expressions, such as the table in Fig. 9 which includes the macro in Fig. 10.

For smaller problems – small tables with no arithmetic expressions – PVS has provided satisfactory performance. For example, PVS found the inconsistency in Effective-SL described in Fig. 13 in approximately 10 s.[4] As a comparison, our approach using BDDs needed only a fraction of a second for the same problem.

For more complex problems – small tables with arithmetic expressions – PVS eliminates all our problems with spurious errors while providing adequate performance. For an example involving two transitions with guarding conditions containing 8 predicates each (12 of which contained arithmetic expressions), it took PVS under 8 s to prove that the transitions did not conflict. On the other hand, abstracting to propositional logic and using our BDD approach required only a fraction of a second but generated 37 spurious error reports that had to be eliminated through time-consuming manual inspection.

[4] Seven subgoals were generated which reduced to one subgoal representing the inconsistency.

For problems in the last class the abstraction to propositional logic starts becoming unsatisfactory; the number of spurious error reports starts to be unmanageable. Manual inspection is a time-consuming and error-prone process and the increased accuracy introduced by PVS is much needed.

For the most complex parts of TCAS II, for example, transitions with guarding conditions the size of the condition in Fig. 9, the propositional logic abstraction approach is clearly unsatisfactory (as described in Section 4.2). The time needed for this analysis is acceptable (3.5 s for the example described in Section 4.2), but the shear size of the error reports (most of which we later showed to be spurious) is too large to report to the analyst and too large for the analyst to feasibly interpret.

Our early attempts to use PVS on these most complex problems were not completely successful. We were unable to complete any proof in PVS, and notified SRI International regarding our problem. Sam Owre of SRI International found that the BDD package PVS uses for first reduction of the proof discovers in a second or two that the particular test case we inquired about is not a tautology [34]. PVS, however, calls the BDD package a second time to convert the formula to a minimal representation to produce the fewest, simplest subgoals. The second call was not terminating since there were several thousand subgoals. An updated release of PVS was modified so the system would not attempt the minimization when there are large numbers of subgoals. The proof attempt using the updated release of PVS on the particular guarding conditions SRI worked with to identify the problem, successfully terminated after approximately 3 h[5] and yielded 1129 unprovable subgoals (each unprovable subgoal represents a potential inconsistency in the specification of the guarding conditions). Unfortunately, for other large guarding conditions involving the `Threat` macro,[6] PVS still ran on the order of days without generating any results, and eventually we aborted the proof attempts. We do not know the exact reasons for these failed proof attempts, however, subsequent research [12] revealed that information related to the structure of the state machine model of the system was missing from the analysis model. Since our analysis is local, we only investigate the transitions out of a single state. We do not take into account global invariants that hold in a particular state. In our work we detected the invariants manually with mechanized decision support [12]. Once this information was identified and incorporated into the analysis model, the PVS proofs successfully terminated and showed the guarding conditions consistent.

Automatically extracting invariants from the specification is an active research area [5]. Integration with an invariant detection tool would have been greatly beneficial. Again, the work described in this paper inspired research in the area of invariant generation, and a prototype tool performing automatic invariant extraction on RSML specification has been developed [29].

[5] The proof was run on a Ross-based SPARC 20 with two 150 MHz CPUs. The time to complete the proof is dependent on the system load. The time reported was the fastest time; a subsequent run ran for a little over 9 h.

[6] The `Threat` macro is one of the most complex macros in the TCAS II requirements specification.

5.4. Comments on PVS

PVS has powerful theorem proving capabilities and largely solves our problems with spurious error reports. The large prelude file contains many built-in theories, and the decision procedures for equality and linear inequality (that are complete for linear arithmetic on the reals) give PVS a significant advantage over theorem provers without these two features. In particular, for our purposes, the decision procedures are extremely useful. Rushby notes that "it is enormously tedious to perform verifications involving even modest quantities of arithmetic in systems such as HOL that lack decision procedures" [33]. Although the decision procedures mainly deal with linear arithmetic, they can also help with some simple nonlinear reasoning, but the decision procedures are not complete for this case [32].

The theorem prover in PVS is an interactive theorem prover and so it requires the user to guide the prover through the proof process. Given that one of our goals is that the analysis procedure be automated and easy to use for personnel with no formal training, we perceived the interactive nature of PVS to be a detriment. However, PVS allows the analyst to define strategies that use a single command to carry out a sequence of steps in the proof process. The relative simplicity of our proofs makes it easy to create customized strategies that work well in most cases. The examples cited in this paper were all proved with a single strategy to check for completeness and a single strategy to check for consistency. By providing a small set of strategies suitable for the types of proofs we want to perform, the proof process can be simplified to a point where little training is needed to use the tool.

6. Related work

The software specification static analysis research related to the work described in this case study generally falls into two categories: (1) completeness and consistency analysis and (2) model checking. We will briefly discuss this related work in this section.

6.1. Completeness and consistency analysis

Heitmeyer et al. [19] define consistency checking of an SCR specification to be a combination of (1) simple syntactic and semantic checks such as type correctness, (2) *coverage*, and (3) *disjointness*. Disjointness in SCR means that each table defines a deterministic relation. Coverage means that each condition table completely defines the relation. Taken together, these criteria ensure that the specification is a total function.

The analysis discussed in this paper addresses two properties, which we call *consistency* and *completeness*. Naturally, an RSML specification is expected to be syntactically correct and satisfy all constraints typically enforced by a strict compiler, such as

type correctness. A consistent specification is one where for each state there is at most one transition that can be taken for any specific event. In a complete specification there must be at least one transition out of each state for any specific event. Thus, disjointness in SCR and consistency in RSML refer to the same concept, and coverage in SCR and completeness in RSML are also similar concepts.

In the SCR analysis tool (called the verifier) they use structural equivalence to determine if two predicates are the same. Instead of BDDs (as was done in our initial tool) they use a tableaux-based decision procedure for propositional logic as defined by Smullyan [35]. Thus, the SCR analysis tool suffers from the same drawbacks and problems with spurious errors as our initial BDD implementation.

6.2. Model checking

Model checking has traditionally been applied to hardware verification. Recently, several groups have investigated applying model checking to software systems.

Atlee and Gannon [3, 2] have applied model checking to the requirements of a cruise control system and a water-level monitoring system and showed how model checking could be used to verify safety properties for event-driven systems.

More recently, Sreemani and Atlee [39] presented a case study of model checking the non-trivial A-7E requirements document for specific properties that the system requirements should satisfy. The case study is intended to demonstrate the scalability of model checking software requirements. They implemented a program that translates an SCR requirements specification into an equivalent Symbolic Model Verifier (SMV) [25] specification, and use SMV to verify the required properties. The model is an abstraction of the original specification. The authors state that

> *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.*

The authors further state that the SCR methodology requirements that must be satisfied, ultimately provide an easy method for obtaining abstractions; i.e., since the SCR methodology already requires abstractions, it is easy to determine additional abstractions for the SMV model. Most of the conditions are represented as Boolean variables, but if two or more conditions are related such that only one and exactly one condition can be true at all times, the conditions are represented as enumerated-type variables [39].

Anderson et al. have investigated the feasibility of model checking large software specifications [1]. They translated a portion of the TCAS II requirements specification into a form acceptable to a model checker and used the model checker to analyze for a number of properties of the system.

Most recently, Bharadwaj and Heitmeyer [6] have integrated the Spin model checker [22] into the SCR toolset so users can establish logical properties of an SCR

specification. To limit the state explosion, they verify abstractions of the original requirements. They describe two types of reductions (abstractions) that are derived using the formula to be verified and special attributes of SCR specifications. The first reduction involves eliminating irrelevant entities in the original model; irrelevant entities are entities that are not needed in the analysis since they do not appear in the formula being checked. The second reduction involves using more abstract representations of monitored variables. Essentially, the abstraction involves eliminating certain monitored variables from the identified entity set that are causing state explosion. The reductions are applied manually and the abstract model produced is then analyzed automatically by the SCR toolset using Spin.

Since all of the methods described in this section rely on various abstractions to generate a system model that can be analyzed in a computationally tractable manner (i.e., avoid the state explosion problem), spurious errors may be reported that would be eliminated if certain abstractions were not made. Currently, it is left up to the analyst to determine which error reports represent true errors and which error reports represent spurious errors.

7. Conclusion

Our effort to analyze the TCAS requirements has achieved two goals. First, we have demonstrated that static completeness and consistency analysis of a complex software requirements specification is both feasible and effective [18, 23].

Second, the size and complexity of the TCAS II requirements specification helped us identify some weaknesses in our original approach and made us recognize the need for increased accuracy in our analysis. In our opinion, it is not feasible to reduce the number of spurious error reports by limiting the expressive power of the modeling language; to make our modeling approach accessible to industrial users we want to maintain the usability and expressive power of the language. There are two main ways of increasing the accuracy of the analysis; (1) provide more precise abstraction functions or (2) provide more powerful decision procedures. To increase the accuracy of our analysis we chose to pursue the latter. We did, however, come to the conclusion that in the interest of time and effort it was more cost effective to integrate with an existing tool providing the analytical power we needed. We chose to work with the Prototype Verification System (PVS).

We augmented our existing tool to generate proof obligations in the PVS specification language and evaluated the approach on examples from the TCAS II requirements specification. The results of the investigation have been largely positive. During the investigation we made four main observations: (1) the theorem proving component of PVS is powerful and can solve many of our problems with spurious error reports, (2) the PVS specification environment and theorem prover are relatively easy to use and have many features that provide advantages over other stand-alone theorem provers, (3) for most test cases, PVS performed efficiently (in terms of execution time) and was

highly automatable, requiring only one user-defined command (strategy) to complete the proofs, and (4) for the largest test cases, the efficiency of PVS degraded to a point where the tool did not terminate.

From our experiences with the development and analysis of the TCAS II requirements specification we offer the following observations.

The ease of use and expressive power of RSML were major factors in the success of the TCAS II project. The participants could capture the right information in the way they felt the information should be captured. This flexibility and power is, in our opinion, essential if a specification approach is going to be widely used and accepted. Unfortunately, the flexibility and power of the language complicates the analysis process.

During our analysis work, we found that the common abstractions used in many tools, for example, not interpreting certain functions such as inequalities and arithmetic expressions, in some cases can lead to excessive numbers of spurious errors. To fully analyze a system such as TCAS II for completeness and consistency one must have access to the reasoning power of a theorem prover.

These observations help us identify promising avenues for continued work in static analysis of state-based software requirements models. From our perspective, areas that would have a significant impact on the state of the art in software specification analysis are integrative analysis methods and integrated analysis tools.

Integrative analysis methods are approaches to static analysis where a combination of tools are used to achieve some goal. In our work we have found that one tool never satisfies the analysis needs, not even in the limited domain of completeness and consistency analysis. In many instances abstraction to propositional logic and a simple BDD-based tool worked fine, in some instances the full power of a theorem prover was suitable, and, finally, in a small number of cases extensive human ingenuity was needed to complete the analysis. Toolsets and analysis approaches integrating various classes of analysis tools would provide the analytical power as well as the level of automation necessary to make static analysis of software specifications widely accepted in industrial settings. An extensible framework in which tool fragments such as decision procedures and abstraction mechanisms can be easily integrated would, in our pragmatic opinion, be a major contribution.

In addition, defined processes and heuristics that can help guide an inexperienced analyst through a verification effort are also needed. We have begun work in this area [12], where we have developed a process for integrating our existing completeness and consistency analysis tools and developed a set of heuristics to help guide the analyst through the verification process [13].

In closing, formal modeling and automated analysis of large commercial software specifications are clearly feasible. Nevertheless, much work remains to be done, particularly in two areas: (1) integration of tools and methods to take advantage of their strengths and circumvent their weaknesses and (2) development of processes and heuristics that can help inexperienced users to easily perform advanced static analysis of large software specifications.

References

[1] R.J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, J.D. Reese, Model checking large software specifications, in: D. Garlan (Ed.), Proc. 4th ACM SIGSOFT Symp. on the Foundations of Software Engineering (SIGSOFT' 96), October 1996.

[2] J.M. Atlee, M.A. Buckley, A logic-model semantics for SCR software requirements, in: S.J. Zeil (Ed.), Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA' 96), January 1996, pp. 280–292.

[3] J.M. Atlee, J. Gannon, State-based model checking of event-driven system requirements, IEEE Trans. Software Eng. 19 (1) (1993) 24–40.

[4] C. Barrett, D. Dill, J. Levitt, Validity checking for combinations of theories with equality, in: M. Srivas, A. Camilleri (Eds.), Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science, vol. 1166, Springer, Berlin, November 1996, pp. 187–201. Palo Alto, CA, November 6–8.

[5] S. Bensalam, Y. Lakhnech, H. Saidi, Powerful techniques for the automatic generation of invariants, in: R. Alur, T.A. Henzinger (Eds.), Proc. 8th Int. Conf. on Computer Aided Verification, CAV' 96, Lecture Notes in Computer Science, vol. 1102, Springer, Berlin, August 1996, pp. 323–335.

[6] R. Bharadwaj, C. Heitmeyer, Verifying SCR requirements specifications using state exploration, in: 1st ACM SIGPLAN Workshop on Automatic Analysis of Software, January 1997.

[7] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. C-35(8) (1986) 677–691.

[8] M. Chechik, J. Gannon, Automatic verification of requirements implementations, in: Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis, 1994, pp. 1–14.

[9] M. Chechik, J. Gannon, Automatic analysis of consistency between implementations and requirements: a case study, in: Proc. 10th Annual Conf. on Computer Assurance, COMPASS 95, 1995, pp. 123–131.

[10] E.M. Clarke, O. Grumberg, D.E. Long, Model checking and abstraction, ACM Trans. Programm. Lang. Systems 16(5) (1994) 1512–1542.

[11] J. Crow, S. Owre, J. Rushby et al., A tutorial introduction to PVS, in: WIFT 95: Workshop on Industrial-Strength Formal Specification Techniques, 1995.

[12] B.J. Czerny, Integrative analysis of state-based requirements for completeness and consistency, Ph.D. Thesis, Michigan State University, May 1998.

[13] B.J. Czerny, M.P.E. Heimdahl, Automated integrative analysis of state-based requirements, in: Proc. 13th IEEE Int. Conf. on Automated Software Engineering (ASE'98), Honolulu, Hawaii, October 1998, pp. 125–134.

[14] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Programm. (1987) 231–274.

[15] D. Harel, On visual formalisms, Commun. ACM 31 (5) (1988) 514–530.

[16] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, Statemate: a working environment for the development of complex reactive systems, IEEE Trans. Software Eng. 16(4) (1990) 403–414.

[17] D. Harel, A. Pnueli, On the development of reactive systems, in: K.R. Apt (Ed.), Logics and Models of Concurrent Systems, Springer, Berlin, 1985, pp. 477–498.

[18] M.P.E. Heimdahl, N.G. Leveson, Completeness and consistency analysis of state-based requirements, IEEE Trans. Software Eng. TSE-22(6) (1996) 363–377.

[19] C.L. Heitmeyer, R.D. Jeffords, B.G. Labaw, Automated consistency checking of requirements specifications, ACM Trans. Software Eng. Methodol. 5(3) (1996) 231–261.

[20] C.L. Heitmeyer, B.L. Labaw, D. Kiskis, Consistency checking of SCR-style requirements specifications, in: Proc. 2nd IEEE Int. Symp. on Requirements Engineering, March 1995.

[21] K.L. Heninger, Specifying software requirements for complex systems: new techniques and their application, IEEE Trans. Software Eng. 6(1) (1980) 2–13.

[22] G.J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[23] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, J.D. Reese, Requirements specification for process-control systems, IEEE Trans. Software Eng. (1994) 684–706.

[24] N.G. Leveson, M. Heimdahl, H. Hildreth, J. Reese, TCAS II requirements specification.

[25] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, Dordrecht, 1993.

[26] S.P. Miller, M. Srivas, Formal verification of the AAMP5 microprocessor, in: Proc. Int. Workshop on Industrial Strength Formal Techniques, 1995, pp. 2–17.

[27] S. Owre, N. Shankar, J.M. Rushby, The PVS Specification Language, Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, April 1993.
[28] S. Owre, N. Shankar, J.M. Rushby, User Guide for the PVS Specification and Verification System, Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993.
[29] D.Y.W. Park, J.U. Skakkebæk, D.L. Dill, Static analysis to identify invariants in RSML specifications, in: Formal Techniques in Real-Time and Fault Tolerant Systems, Lyngby, Denmark, September 1998.
[30] D.Y.W. Park, J.U. Skakkebæk, M.P.E. Heimdahl, B.J. Czerny, D.L. Dill, Checking properties of safety critical specifications using efficient decision procedures, in: Proc. FMSP' 98: Workshop on Formal Methods in Software Practice, March 1998.
[31] W. Pugh, A practical algorithm for exact array dependence analysis, Commun. ACM 35(8) (1992) 102–114.
[32] J.M. Rushby, personal communication, 1995.
[33] J. Rushby, Formal specification and verification for critical systems: Tools, achievements, and prospects, in: Electric Power Research Institute (EPRI) Workshop on Methodologies for Cost-Effective, Reliable Software Verification and Validation, January 1992, pp. 9–1 to 9–14.
[34] J.M. Rushby, F. von Henke, Formal verification of algorithms for critical systems, IEEE Trans. Software Eng. 19(1) (1993) 13–23.
[35] R.M. Smullyan, First-Order Logic, Springer, New York, 1968.
[36] N. Shankar, Mechanical verification of a schematic protocol for byzantine fault-tolerant clock synchronization, Technical Report SRI-CSL-91-04, SRI International, June 1991.
[37] N. Shankar, S. Owre, J.M. Rushby, The PVS Proof Checker: A Reference Manual, Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993.
[38] R.E. Shostak, A practical decision procedure for arithmetic with function symbols, J. ACM 26(2) (1979) 351–360.
[39] T. Sreemani, J.M. Atlee, Feasibility of model checking software requirements, in: 11th Annual Conf. On Computer Assurance COMPASS, June 1996, pp. 77–88.
[40] P. Wolper, B. Boigelot, An automata theoretic approach to presburger arithmetic constraints, in: Proc. of Static Analysis Symp., Lecture Notes in Computer Science, vol. 983, Springer, Berlin, September 1995, pp. 21–32.
[41] J. Yang, A.K. Mok, F. Wang, Symbolic model checking for event-driven real-time systems, ACM Trans. Programm. Lang. Systems 19(2) (1997) 386–412.