Completeness and Consistency in Hierarchical State-Based Requirements

Mats P.E. Heimdahl and Nancy G. Leveson

Abstract—This paper describes methods for automatically analyzing formal, state-based requirements specifications for some aspects of completeness and consistency. The approach uses a low-level functional formalism, simplifying the analysis process. State-space explosion problems are eliminated by applying the analysis at a high level of abstraction; i.e., instead of generating a reachability graph for analysis, the analysis is performed directly on the model. The method scales up to large systems by decomposing the specification into smaller, analyzable parts and then using functional composition rules to ensure that verified properties hold for the entire specification. The analysis algorithms and tools have been validated on TCAS II, a complex, airborne, collision-avoidance system required on all commercial aircraft with more than 30 passengers that fly in U.S. airspace.

Index Terms—Completeness, consistency, static analysis, reactive systems, state-based requirements, formal semantics, formal methods.

1 Introduction

A software requirements specification should be a comprehensive statement of a software system's intended behavior. Unfortunately, requirements specifications are often incomplete, inconsistent, and ambiguous. We know that many serious conceptual errors are introduced in this first stage of software development—errors introduced during the requirements stage have been shown to be more difficult and more expensive to correct than errors introduced later in the lifecycle, and they are more likely than implementation errors to be safety critical [24], [25]. Therefore, it is important to provide methods and techniques to eliminate requirements-related errors as early as possible.

To provide analysis procedures to find errors in specifications, it is first necessary to determine the desirable properties of a specification. Previously, we defined formal criteria for requirements completeness, consistency and safety. Jaffe, in his dissertation, defined a rigorous basis for ascertaining whether or not a given set of software requirements is internally complete, i.e., closed with respect to statements and inferences that can be made on the basis of information included in the specification [21]. Emphasis is placed on aspects of requirements specification that are usually not adequately handled, including timing and robustness, and on aspects that are particularly related to safety and accidents.

The definition of specification completeness provided by Jaffe was subsequently formalized using a simple Mealy-machine model called RSM (Requirements State Machine) [20]. The RSM notation was developed solely as a means for formally defining our criteria and lacks most desirable properties of a true requirements specification language. To be useful in practical applications, these criteria need to be translated into criteria applicable to a real specification language. Although the criteria could be applied to many languages, we chose to work with a formal, state-based specification language called RSML (Requirements State Machine Language). RSML was developed by the Irvine Safety Research Group using a real aircraft collision-avoidance system called TCAS II (Traffic alert and Collision Avoidance System II) as a testbed [23].

This paper defines the formal semantics of RSML and describes an automated approach to analyzing an RSML specification for two qualities: 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 nondeterminism). The need for consistency is obvious, but the robustness criteria require further explanation.

Embedded software 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 [11]. A *robust* system will detect and respond appropriately to violations of assumptions about the system environment (such as unexpected inputs). Robustness with respect to a state-machine description implies the following:

1) Every state must have a behavior (transition) defined for every possible input.
2) The logical OR of the conditions on every transition out of any state must form a tautology.
3) Every state must have a software behavior (transition) defined in case there is no input for a given period of time (a timeout).

Thus, the software must be prepared to respond in real time to all possible inputs and input sequences. That is, the

- *M.P.E. Heimdahl is with the Department of Computer Science, University of Minnesota, 4-192 EE/CS Building, 200 Union Street S.E., Minneapolis, Minnesota 55455-0159. E-mail: heimdahl@cs.umn.edu.*
- *N.G. Leveson is with the Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195. E-mail: leveson@cs.washington.edu.*

Manuscript received June 1995; revised January 1996.
Recommended for acceptance by D. Notkin and D.R. Jeffery.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S96048.

software must be complete with respect to its input domain. In the rest of this paper, we use the term d-complete to represent this aspect of requirements completeness.

Manually verifying compliance with our set of criteria is a time-consuming and error-prone process. Thus, tools that support automated verification would be highly desirable. This analysis, unfortunately, is computationally expensive and infeasible in most specification languages. To overcome this problem, the semantics of RSML was defined with *analyzability* as one of the main goals.

In order to accomplish our goal of analyzability, we view a specification expressed in RSML as a mathematical relation composed from simple, analyzable parts. The compositionality is achieved through the definition of the next-state relation and by enforcing some simple restrictions on the way in which a system can be modeled. The compositional approach allows us to partition a large problem into small manageable pieces, perform the analysis on each separate piece, and then combine the individual analysis results into a statement about the entire system. Analysis procedures that are too costly to apply to the monolithic problem can then be applied to manageable subsets of the problem and the individual results combined to make a statement about the original problem.

Related approaches to requirements analysis include methods based on formal proof systems and different static analysis techniques such as reachability analysis and model checking.

Formal Proof Systems. Formal proof systems can be powerful tools in the verification of critical properties of algorithms [29]. Attempts have been made to extend the use of formal proofs and apply them to requirements specifications, for example, the ProCoS (Provably Correct Systems) project [27], [28]. Unfortunately, the languages used in the theorem proving approach, such as process algebras and higher order logics, are not understandable by the non-software professionals involved in most requirements specification efforts and thus are not (in our opinion) suitable as high-level requirements languages. Also, formal proofs are notoriously difficult to derive, and these approaches may not be practical for complex systems.

Reachability Analysis. Modeling a system as a finite-state machine and then performing reachability analysis of the global state space has been successfully used in the analysis of communication protocol specifications [8], [19], [18]. The main problem with reachability analysis is that it relies on the generation of a global reachability graph and, therefore, quickly runs into a state-space explosion problem.

Model Checking. Model checking is conceptually simple and is applicable in a wide variety of languages and application areas [1], [6], [7]. Early work in model checking also relied on a global reachability graph. Consequently, the approach suffered from state-space explosion problems. Newer approaches relying on a symbolic representation of the state space can significantly improve the performance of the model checking approach [5]. Symbolic model checking has been applied to large models [5], [4], but only for systems with simple, repetitive elements—such as those commonly found in hardware applications. The time and space complexity of the symbolic approach is affected not only by the size of the specification but also by the regularity of specification. Software requirements specifications lack this necessary regular structure, and it is unclear how well the symbolic approach will perform on these specifications.

Our approach differs from these techniques in that it performs the analysis directly on a high-level requirements model without generating a global reachability graph. Thus, the analysis is both conceptually simple and eliminates the problem with state-space explosion.

Recently, Heitmeyer, Labow, and Kiskis have published a paper [15] discussing some aspects of consistency and completeness in the context of SCR-style (Software Cost Reduction [16], [17]) requirements specifications. SCR is a state-based approach using an assortment of tabular notations to define state transitions (or mode transitions as they are called in SCR) and output variables. The consistency checks described in [15] are concerned with language properties such as proper syntax of the specification and type correctness, as well as a notion of local consistency and completeness of individual tables. The latter notion of consistency of tables is similar to the completeness and consistency properties we are investigating in this paper. However, their approach investigates if one table is internally consistent and does not provide a statement about the system as a whole.

To ensure that the formal RSML specification language and the associated analysis algorithms and tools are appropriate for large and realistic systems, a testbed specification was developed for TCAS II [22]. The testbed is currently being used to develop and validate various types of analysis algorithms and tools on the underlying formal model. TCAS II has been described by the head of the TCAS program at the FAA as the most complex system to be incorporated into the avionics of commercial aircraft. It therefore provides a challenging experimental application of formal methods to a real system.

This paper documents our approach to static analysis of RSML and gives examples of the types of problems that d-completeness and consistency analysis are capable of detecting. Section 2 gives a short introduction to the features of RSML necessary to understand this paper. Section 3 provides a formal definition of these RSML features. The definition is based on the notion that a transition in a simple state machine can be viewed as a function mapping the current state to the next state and the behavior of a hierarchical state machine can be viewed as a composition of simple functions. Automated analysis procedures for d-completeness and consistency are outlined in Section 4. An evaluation of the algorithms and examples of the types of problems this analysis is capable of detecting is described in Section 5. Section 6 presents conclusions.

2 Overview of the RSML Notation

RSML is a state-based requirements specification language suitable for the specification of reactive systems. RSML includes several features developed by Harel for Statecharts [9], [10]: superstates, AND decomposition, broadcast communication, and conditional connectives. In addition,

RSML has some unique syntactic and semantic features that were developed to enhance readability, reviewability, and analyzability and our ability to handle complex systems.

A complete description of RSML is provided in [23]. This section contains only a description of the RSML features necessary to understand this paper.

A simple finite-state machine is composed of *states* connected by *transitions* (see Fig. 1). *Default* or start states are signified by states whose connecting transition has no source. In Fig. 1, state *A* is the start state. Transitions define how to get from one state to another. In Fig. 1, states *B* and *C* are directly reachable from *A*. State *D* is only indirectly reachable from *A* via state C.

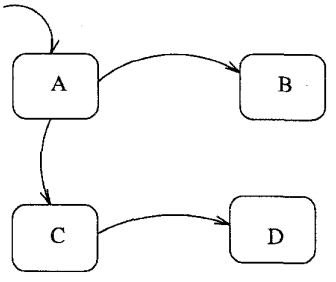

Fig 1. A basic state machine.

Superstates. In RSML (and Statecharts), states may be grouped into *superstates* (see Fig. 2). Such groupings reduce the number of transitions by allowing transitions to and from the superstate rather than requiring explicit transitions to and from all of the grouped states (*substates*). Superstates can be entered in two ways. First, the transition to the superstate may end at the superstate's border (transition *A* in Fig. 2). In this case, a default state must be specified within the superstate. In the example, state *S* is entered upon taking transition *A*. Alternatively, the transition may be made to a particular state inside the superstate (transition *B* in Fig. 2). The same superstate may have transitions ending at the border and at any number of the inner states. The superstate may be exited in two ways (transitions C and *D* in Fig. 2). Analogous to transitions into the superstate, transitions out of the superstate may originate from the border or from an inner state. The same superstate may contain both types of exiting transitions. Note that all transitions to and from the superstate boundary can be redrawn to cross the boundary and enter the substates explicitly (Fig. 3).

AND Decomposition. One of the most important innovations in Statecharts is what Harel calls an *orthogonal product*,[1] which contains two or more parallel state machines. In RSML these states will be referred to as parallel states and are indicated by a gray background (Fig. 4). When the parallel state *S* is entered, *each* of the state machines *A*, *B*, *C*, and *D* within it is entered. All state machines are exited when *any* transition is taken out of the parallel state. The use of parallel states greatly reduces the size of the specification. For example, we estimate that TCAS (i.e., the complete reachability graph) contains at least 10^{40} states, whereas the hierarchical state diagram in our RSML specification of TCAS has approximately 140 states and fits on five pages.

1. Orthogonal products are also known as "parallel states," "product states," and "AND states."

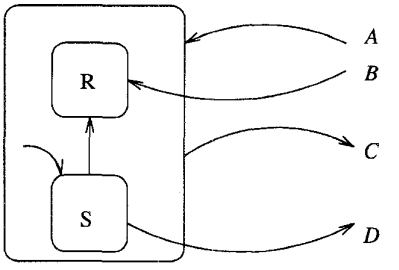

Fig. 2. A superstate example.

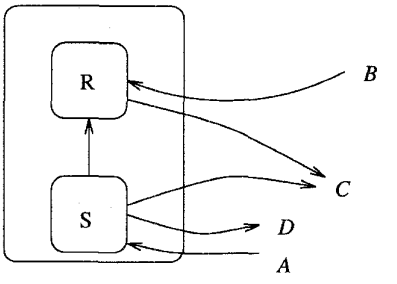

Fig. 3. Transitions redrawn to bypass the superstate.

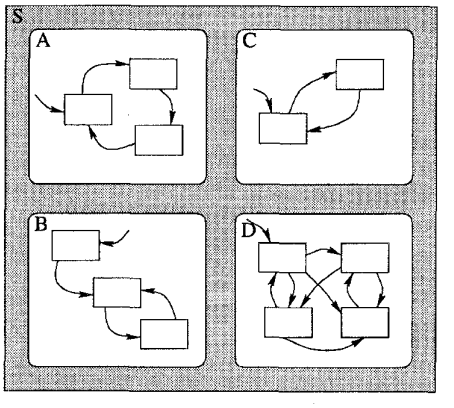

Fig. 4. The parallel state.

Transition Definitions. Transition definitions in RSML contain five parts: 1) the identification (the source and destination of the transition), 2) the location (the state machine in which the transition is located), 3) the triggering event, 4) the guarding condition, and 5) the output action. The identification, location, and triggering event are the only required parts. Fig. 5 shows the syntax of a transition definition in RSML.

Transitions are taken upon the occurrence of the *trigger event*, provided that the guarding condition is true. The *guarding condition* defines preconditions on the transition and is specified using AND/OR tables, described below. *Output actions* identify events that are generated when the transition is taken. These newly generated events may now trigger transitions elsewhere in the state machine.

Although specification languages such as Statecharts and RSML can be used for many purposes, RSML was explicitly designed to be used for pure black-box requirements specifications. Such specifications describe only the externally visible behavior of the system component being defined in terms of a model of the relationship (mathematical relation) between the inputs and outputs. In addition, RSML specifications describe this behavior (relation) only in terms of variables and conditions of objects *external* to the computer (the sensors, actuators, and system components controlled by the software).

Therefore, internal events in RSML specifications are used only for one very specific purpose: to order the evaluation of the mathematical (input/output) relation to be computed by the software. Basically, they serve the same purpose as parentheses in algebraic equations. Viewing an RSML specification as a mathematical relation is the basis for our formalization of the language and will be described in detail in Section 3.

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	
AND	Auto-$SL_{s\text{-}30}$ **in state** ASL-2	T	T	·
	Auto-$SL_{s\text{-}30}$ **in one of** {ASL-2,ASL-3,ASL-4,ASL-5,ASL-6,ASL-7}	·	·	T
	Lowest-$Ground_{f\text{-}241} = 2$	·	·	T
	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.

AND/OR Tables. Statecharts use predicate calculus to describe the guarding conditions on the transitions [2], [9]. Our TCAS external reviewers (including avionics engineers, component engineers, airline representatives, and pilots), however, did not find this notation natural or reviewable. Instead, we decided to use a tabular representation of disjunctive normal form (DNF) that we call AND/OR tables (see 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 are true. A dot denotes "don't care."

The next section formally defines the structure of these basic syntactic features and gives a formal definition of the semantics of RSML based on the composition of mathematical functions.

3 A Functional Framework

The behavior of a finite-state machine can be formally defined using a next-state relation. In RSML, this relation is modeled by transitions and the sequencing of events. Thus, one can view a graphical RSML specification as the definition of the mathematical next-state *relation F*. If, however, the relation F behaved as a mathematical *function*, and it was defined over all possible system states, some highly desirable properties of requirements specifications would be satisfied:

- The model M would have a response specified for every possible input (i.e., it would be d-complete),
- The model would have no conflicting requirements (i.e., it would be consistent), and
- The model would be deterministic.

Thus, by forcing the behavior of an RSML specification to be a mathematical function, we can guarantee the d-completeness, consistency, and determinism of a requirements specification. This is really the essence of the difference between our approach and others. Instead of allowing the next-state relation to be defined in a way that makes the analysis procedures difficult and then working hard to find analysis procedures that will work on the resulting model, we limit F (the next-state relation) in the language semantic definition in a way that makes the analysis relatively easy. Our resulting analysis algorithms are simple and can be performed directly on the model without needing to generate any part of the reachability graph.

As a side benefit, during the TCAS specification development we found that this next-state relation was easier for the reviewers to interpret correctly than the alternatives we tried (it seemed to satisfy their intuitive understanding of state machines better). In fact, we decided on this semantic definition before we discovered that it simplified the analysis. Perhaps this just confirms the hypothesis that has been occasionally raised that languages for which the formal semantic definitions are simple also seem to be the easiest for users to understand and use correctly.

It may seem overly restrictive to require that the behavior of the software be limited to a mathematical function. However, safety-critical software should not be incompletely specified. In [20], we define requirements *completeness* as the specification being sufficient to distinguish the behavior of the desired software from that of any other, undesired program that might be designed. Nondeterministic specifications often hide dangerous incompleteness in this sense. In this paper, we show a nondeterminism in the TCAS specification we found that was unplanned, had serious safety implications, and was not obvious to us when developing the specification. If one of several possible alternatives is preferable with respect to some desired system quality, then this decision needs to be made by application experts, not by the programmers or software engineers, and it should be made during the requirements analysis process rather than later. Identifying nondeterminism in the specification will help with this decision-making. If two behaviors are identical with respect to all desired system qualities (which is highly unlikely), there is still the problem of determining this equivalence. In most cases, it is easier to evaluate a single deterministic behavior for all desired qualities than to evaluate multiple behaviors for all required qualities. In addition, nondeterministic behavior is usually undesirable with respect to the human-machine interface.

This section provides a formal definition of the semantics of the basic features of RSML. Section 3.1 defines the static structure of the state hierarchies. Section 3.2 describes how the dynamic behavior defined by the transitions and events in RSML can be viewed as compositions of functions. Those readers primarily interested in the use of the analysis tools and not in the formal foundation might skip to Section 4.

3.1 Hierarchical State-Machines

An RSML state machine M can be described by a six-tuple: $M = (S, \leq, \sim, V, c_0, F)$ where:

S is a finite set of *states*. These states are used to model the *global system states*. It is important to note the difference between the elements of S (states) and the *global state*. Certain subsets of S, called configurations, represent consistent combinations of states, and the global state of the system contains a configuration as one of its components. *Config*, the set of all configurations, is defined in Appendix A.

$\leq$ is a tree-like partial ordering with a topmost point (called the *root*). This relation defines the hierarchy relation (or parent/child relation) on the states in S ($x \leq y$ meaning that x is a descendant of y, or x and y are equal). Tree-like means that $\leq$ has the following property:

$$\neg(a \leq b \vee b \leq a) \Rightarrow \neg\exists\, x\, :\, (x \leq a \wedge x \leq b)$$

In the graphical notation, this relation is visualized as containment (states are contained within superstates). In Fig. 6, for example, B $\leq$ A, G $\leq$ A, I $\leq$ E, etc.

If the state x is a descendant of y ($x < y$), and there is no z such that $x < z < y$, we say that the state x is a *child* of y (x *child* y). For example, in Fig. 6, the state B is a child of A and H is a child of E.

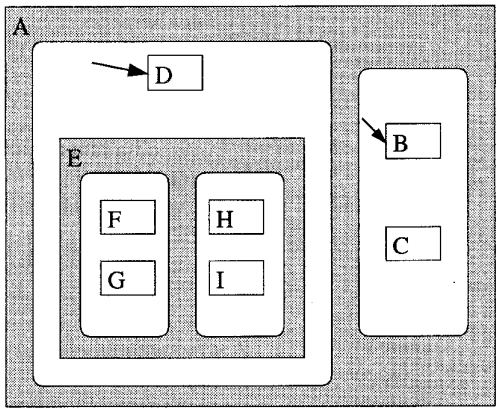

Fig. 6. A sample state hierarchy.

Furthermore, we define $\sigma(y)$ as the set of all children of the state y, that is,

$$\sigma(y) = \{x \mid x \text{ child } y\}$$

$\sim$ is an equivalence relation on the states in $S - \{root\}$ that satisfies one additional property: whenever $x \sim y$, then x and y have the same parent.

$$x \sim y \Rightarrow \exists z\, :\, x, y \in \sigma(z)$$

The equivalence classes in $\sim$ are called *parallel components*. If a state z has two (or more) inequivalent children, then z is said to be a *parallel state* and the *parallel components* of z are the equivalence classes of its children.

The equivalence relation $\sim$ is used to partition the children of a state into disjoint sets. In Fig. 6, for example, the children of A are partitioned into two equivalence classes {B, C} and {D, E}.

V is a set containing the input and output histories of the model (the complete variable traces). The set C of global states is a subset of (*Config* $\times$ V).

c_0 is the initial *global state* of the machine, $c_0 \in$ (*Config* $\times$ V). A global state is an ordered pair consisting of a set of states, called the *configuration* of the machine, and a trace from V. The initial global state in Fig. 6 is defined by the pair ({A, B, D}, $\emptyset$). The properties of a configuration are formally defined in Appendix A.

F is a relation defining the global state changes in the machine M (and the possible changes in the output variables). F is a mapping $C \mapsto C$, where $C \subseteq$ (*Config* $\times$ V). The relation F is also referred to as the *behavior* of M.

In the definitions above, there are few restrictions on the nature of a *global state*. All that is required is that a global state c_i is an element of C. From the discussion in Section 2, it is clear that there are certain combinations of states that are not allowed when describing the global state. In Fig. 6, for example, the states B and C cannot both be part of the global state, that is, the machine cannot be in state B and state C simultaneously. The restrictions governing the structure of a global state have been formally defined for Statecharts by Harel et al. [12]. These definitions are also applicable to RSML. Although the definitions are not essential for understanding the remainder of the paper, for completeness they have been included in Appendix A.

The remainder of this section is devoted to the next-state relation F. We will show how the transitions in RSML can be viewed as mathematical functions and how these functions can be composed to form the complex behavior of the global next-state relation F.

3.2 Next-State Mapping

The hierarchies and parallelism (defined by the functions $\leq$ and $\sim$), together with the definitions in Appendix A, enforce a rigorous structure on the possible global states (the set C). The dynamic behavior (the possible global state changes) is defined by the next-state relation F ($C \mapsto C$). In a model of a system with nontrivial functionality, this mapping will be complex. However, the mapping can be viewed as a composition of smaller, less complex mappings. Specifically, F can be viewed as composed of simple *functions*.

In the graphical notation, these simple functions are defined by transitions. The *domain* of a function is defined by the source, i.e., the state that the tail of the transition is leaving, and the guarding condition on the transition. The *image* of a function is defined by the destination of a transition, i.e., the state the transition enters, and possible output. The functions represented by the transitions are then composed depending on the structure of the particular state machine being considered and the events defined on the transition.

The semantics of RSML are defined using three basic functional compositions:

Union. The union composition of two functions ($g \cup h$) merges the domains of the functions.

DEFINITION 1. *The functional properties are maintained under union* ($\cup$) *iff.*

$$\forall x \in (Dom(g) \cup Dom(h)) \; : \; g(x) = h(x)$$

Union composition of two functions is allowed if the domains of the functions do not overlap, or the domains overlap but the functions are equivalent for all elements in the intersection.

Serial. Serial composition $g(h(c))$ (or $g \circ h$) corresponds to normal functional composition.

DEFINITION 2. *Serial composition* ($g \circ h$) *is allowed iff.*

$$Dom(g) \supseteq Im(h)$$

Informally, serial composition is allowed if the image of the first function applied is a subset of the domain of the second function, i.e., the second function is defined for all possible results of the first function.

Parallel. Parallel application is denoted $\langle h, g \rangle (x)$. Parallelism is modeled as interleaving, i.e., an arbitrary ordering of functional applications.

DEFINITION 3. *Parallel composition is allowed iff.*

$$Dom(g) \supseteq Im(h) \wedge$$
$$Dom(h) \supseteq Im(g)$$

Parallel composition is allowed if both possible serial compositions are allowed. If $g \circ h(x) \neq h \circ g(x)$, i.e., the ordering of the functional application is important, parallel composition will lead to nondeterminism and the properties of a function are lost. The notation $\langle X \rangle$, where X is a set of functions, will be used to denote the parallel composition of the functions in X.

In RSML, union composition occurs between nonparallel transitions triggered by the same event. For example, the functions representing the transitions t_1, t_2, and t_3 in Fig. 7 (assuming all are triggered by the same event) are composed in union.

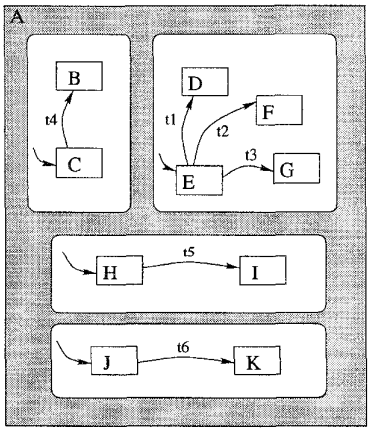

Fig. 7. A sample state machine.

Transitions triggered by the same event, but in parallel state components, are composed in parallel. In Fig. 7, transitions t_3 and t_4 are composed in parallel (assuming they are triggered by the same event).

Finally, serial application is caused by the event propagation mechanism. Assume the transition t_5 is triggered by some external event and generates event e as an action. This event is picked up by transition t_6—that is, t_6 is triggered by e. Thus, transition t_5 is taken first and transition t_6 second. This sequencing is modeled as applying the functions representing t_5 and t_6 in series: $f_{t_6} \circ f_{t_5}(c)$.

In this way the complete behavior of any model can be hierarchically defined as a composition of the behaviors of its parts.

Before we can define the complete behavior of an RSML specification, we have to investigate the nature of the functions defined by the transitions. A function f can be textually defined by

$$f(c, v) = ((c - Q_s) \cup Q_d, v') \textbf{ if } (x \in c) \wedge p(c, v)$$

where Q_s and Q_d are sets of states, v' is an updated variable trace, x is a state, and p is an arbitrary predicate over the global state c. In the graphical representation, this function represents a transition with the tail in the state x and the guarding condition p. If the transition is taken, the structure of the state machine may cause more states than x to be exited—for example, if x is a superstate. The set of states that is exited when the transition is taken is denoted by Q_s and the set that is entered by Q_d. In the definition above, the set $\{(c, v) \mid (x \in c) \wedge p(c, v)\}$ defines the domain of the function ($Dom(f)$), and Q_s and Q_d are the *source* ($Source(f)$) and *destination* ($Dest(f)$) of f, respectively. Note that the domain is defined over the set of global states, that is, $Config \times V$, and the source and destination are sets of states.

The functional definition of a complete RSML specification is recursively built from (composed of) the functional definitions of its components. To define this recursion we need to introduce some auxiliary concepts.

Let E be the set of all events in a model M. A transition is defined by a tuple $((C \mapsto C) \times E \times 2^E)$. The components of the tuple are denoted by *map*, *trigger*, and *actions* respectively. The *map* function is defined as outlined earlier in this section. Let T be the set of all transitions. Furthermore, let $Stages = S \cup \Pi$, where Π is the set of equivalence classes of $\sim$. Also, for any $s \in S$, let $\pi(s)$ be the set of equivalence classes of children of s:

$$\pi(s) = \{x \in \Pi \mid x \subseteq \sigma(s)\}$$

For each $t \subseteq T$ we will define a function

$$g[t] \; : \; Stages \mapsto (C \mapsto C)$$

that defines the behavior of states and parallel components given a set of trigger events. The function g is defined by induction on *Stages* over the relation $\lessdot$ defined as follows:

For all $s \in S$, $p \in \Pi$, and $st \in Stages$

$$s \lessdot st \text{ iff } st \in p \text{ and } s \in st \quad (1)$$

$$p \lessdot st \text{ iff } st \in S \text{ and } p \in \pi(st) \quad (2)$$

Induction over $\lessdot$ is valid because it is well-founded: Whenever $s_1 \lessdot p \lessdot s_2$, it follows that $s_1 < s_2$. Therefore, $\lessdot$ does not contain an infinite descending chain.

The behavior of a composed state (a superstate) is defined as the parallel composition of its parallel state components. In Fig. 7, for example, the behavior of the state A is defined as the parallel composition of its four parallel state components.

DEFINITION 4. *For any $p \in \Pi$ and $t \subseteq T$, the behavior (g) of a state $s \in S$:*

$$g[t]_s = \langle \{ g[t]_p \mid p \lessdot s \} \rangle$$

Informally, one can view the components of a composed state as processes, and the behavior of the composed state as the parallel execution of these processes.

The behavior of a set of states grouped in a parallel state component is defined as the union of 1) the behaviors of the states included in the component and 2) the behaviors introduced by the transitions between states at this level of abstraction. The notation $tr \sqsupset p$ denotes a transition $tr \in T$ introduced in the parallel state component p. In Fig. 7, the transition labeled with t_4 *belongs* to the parallel component $\{B, C\}$.

DEFINITION 5. *A transition tr belongs to the parallel component p of a state s, that is, $p \in \pi(s)$, (denoted by $tr \sqsupset p$)* iff:

$$\exists x \in Source(tr.map) : x \in p$$

DEFINITION 6. *For any $s \in S$ and $t \subseteq T$, the behavior of a parallel state component $p \in \Pi$:*

$$g[t]_p = \left(\bigcup_{m \lessdot p} g[t]_m \right) \cup \left(\bigcup_{tr \in t \wedge tr \sqsupset p} tr.map \right)$$

Informally, a parallel state component behaves either as one of its states, i.e., the state it is currently in, or according to the transitions between the states contained in the component.

Finally, the behavior of a model M under a specific event e can be defined. Let T_e be the set of all transitions, with the trigger $e \in E$, that is,

$$T_e = \{ tr \in T \mid tr.trigger = e \}$$

DEFINITION 7. *The behavior of M under event $e \in E$ is defined as*

$$F^e = g[T_e]_{root}$$

The rules defined above govern the behavior of M under one specific event, i.e., all transitions in the model triggered by this one event are composed according to these rules. The behavior for all individual events in the model can now be modeled the same way. If an event e is generated, the function defined by the behavior under e, i.e., the behavior generated by composing all transitions triggered by e, is applied, and a new system state is calculated. The only remaining part to model is the event propagation mechanism. After a function has been applied and a new system state calculated, a new function is applied based on the output actions on the transitions used to construct the first function. We call the set of events generated as a result of output actions the *yield* of a next state calculation. The following definitions describe how yield is calculated and how the sequence of next state calculations is determined.

First, for any $f \subseteq E$, let

$$T_f = \bigcup_{e \in f} T_e$$

To calculate the *yield*, define the functions

$$\begin{aligned} Yield1 &: T \times C \times 2^E \mapsto 2^E \\ Yield2 &: Stages \times C \times 2^E \mapsto 2^E \\ Yield3 &: 2^E \times C \mapsto 2^E \end{aligned}$$

as follows.

DEFINITION 8. *For $t \in T$ and $x \in C$, and $f \subseteq E$:*

$$Yield1(t, x, f) = \textbf{if } t \in T_f \wedge x \in Dom(t.map) \textbf{ then } t.actions \textbf{ else } \emptyset$$

Yield2 is defined by induction over the relation $\lessdot$ defined previously.

DEFINITION 9. *For any $x \in C$ and $f \subseteq E$:*

If $s \in S$ then

$$Yield2(s, x, f) = \bigcup_{p \lessdot s} Yield2(p, x, f)$$

If $p \in \Pi$ then

$$Yield2(p, x, f) = \left(\bigcup_{s \lessdot p} Yield2(s, x, f) \right) \cup \left(\bigcup_{t \in T_f \wedge t \sqsupset p} Yield1(t, x, f) \right)$$

Given a global state x and a set of events f, the yield of a next state calculation initiated by the events in f is defined as Yield3.

DEFINITION 10. *For $x \in C$ and $f \subseteq E$:*

$$Yield3(f, x) = Yield2(root, x, f)$$

A next state calculation is always started by the arrival of an input. A sequence of function applications will follow. The next function is always determined by the yield of the previous function. This sequence ultimately will be terminated by the application of a function with no yield. Given any global state $x \in C$ and any $e \in E$, define two sequences $x_i \in C$ and $yield_i \subseteq E$ for all $i \geq 0$ as follows:

$$\begin{aligned} x_0 &= x \\ yield_0 &= \{e\} \\ x_{i+1} &= \left\langle \{ F^d \mid d \in yield_i \} \right\rangle (x_i) \\ yield_{i+1} &= Yield3(x_i, yield_i) \end{aligned}$$

The sequence of next state calculations is terminated when $yield_i \neq \emptyset$, and the new global state is x_i. If $yield_i \neq \emptyset$ for all i, the model is ill-formed and the next state calculation will not terminate.

Note that in order for the compositions defined in this section to maintain the properties of a function, all rules for serial and parallel composition defined in the beginning of Section 3.2 have to be followed.

4 Analysis Approach

If the relation F defining the dynamic behavior of the model is a function, then d-completeness, consistency, and deter-

minism are assured. By investigating the fairly simple compositions defining the dynamic behavior of the system, these properties can be evaluated.

The analysis approach is based on the compositional properties of the semantic definition. The base step in the analysis is to assure that transitions out of atomic states do not conflict and never leave the behavior undefined, i.e., the requirements are d-complete with respect to the individual atomic states. In this way, we can guarantee that *if* the model is in this state and an event triggering any transition out of the state is generated, a transition out is defined independently of the global state and the input that has arrived at the model boundary. The rules for union, parallel, and serial composition can then be applied to show that the behavior of the entire hierarchical and parallel machine is d-complete and consistent. That is, by only investigating the functional compositions, we can ensure that the d-completeness and consistency properties verified for a single state are not compromised by hierarchies, parallelism, and event propagation.

This section outlines algorithms analyzing a model for the satisfaction of functional properties.

Union Composition. Union composition requires that the domains of the functions describing the transitions involved in the composition are disjoint, i.e., no two transitions out of the same state can be satisfied at the same time. In addition, functions require that the entire domain is covered. Thus, there must be a satisfiable transition out of every state independent of what input arrives at the model boundary.

The guarding conditions on the transitions triggered by the same event are pairwise compared to see if they are mutually exclusive. Two transitions with guarding conditions that are not mutually exclusive represent conflicting requirements. In addition, if the logical OR of the conditions on all transitions out of the state triggered by the same event does not form a tautology, then there are conditions for which no behavior is specified, i.e., the requirements are incomplete. Tables 1, 2, and 3 outline the data structures and the algorithm used to analyze for d-completeness and consistency of individual states.

In RSML, the conditions are represented by AND/OR tables (Fig. 5). The conditions for state transition in TCAS II are quite complex, resulting in many cases in large tables and requiring costly logical AND and OR operations on the transitions (satisfiability of Boolean functions is known to be an *NP* problem). Our analysis tools use Binary Decision Diagrams (BDDs) [3] for the manipulation of the conditions. BDDs are data structures used to represent Boolean functions in a canonical form. Initially, our main concern was the performance of the AND and OR operations needed to check for mutual exclusion and complete coverage. With BDDs, Boolean formula manipulation can be performed in time linear to the size of the BDD structure. Unfortunately, in the worst case the size of the BDD structure is exponential in the number of terms in the Boolean function it is representing. The BDD approach has provided excellent performance for all examples from TCAS we have tried. Since the BDDs manipulate predicates symbolically, the analysis is conservative and spurious error reports may be generated. This issue will be covered in more detail in Section 5.1.1.

Serial Composition. Serial application of functions arises out of the event propagation mechanisms provided in RSML (and Statecharts). A transition triggered by event e_1 may generate event e_2 as an action, i.e., if e_1 occurs, the transition is taken and e_2 is generated. The event e_2 may now trigger another transition somewhere else in the model. If an event is generated but does not trigger any transition, it is likely that this event was generated in error or that transitions triggered by this event are missing from the requirements. Serial composition of functions requires that the image of the first function is a subset of the domain of the second function. In the graphical model, this requirement implies that if an event is generated, there must always be a transition elsewhere in the model ready to be triggered by this event. All states have a set of transitions enabled (or ready) that can be taken when the model is in a specific state. Using one bottom-up pass over the state hierarchy, all states can be annotated with the transitions enabled in them.

It is also possible to annotate each state with the states that can coexist in the global state description. With this annotation, assuring that all events generated as actions will be used is straight forward.

Parallel Composition. Parallel composition occurs when two (or more) transitions in parallel state machines are triggered by the same event (or events generated simultaneously). If the truth value of the guarding condition of one transition can be affected by a state change caused by a parallel transition, then there exists a possibility of nondeterminism, and the transitions are said to conflict with each other.

TABLE 1
DATA STRUCTURES USED TO REPRESENT THE STATE SPACE

```
struct State{
   String           name;
   State            parent;
   StateList        children;
   TransitionArray  all_trans_out;
}

struct Transition{
   State      source;
   State      dest;
   Condition  cond;     // The guarding condition on
                        the transition
   Event      trigger;
   EventArray actions;
   InfoSet    uses;     // All elements in M this
                        transition depends on
   InfoSet    effects;  // All elements in M this
                        transition effects
}
```

TABLE 2
DATA STRUCTURE FOR CONFLICTING TRANSITIONS

```
struct Conflict{
   Transition trans1;
   Transition trans2;
   Condition cond;
}
```

TABLE 3
D-COMPLETENESS AND CONSISTENCY IN STATE s UNDER EVENT e

```
void Complete_Consistent_Under_e(State state, Event e){
   Condition defined_for;
   defined_for = FALSE;
   for (int i = 0; i < size_of(state.all_trans_out); i++){
      if (state.all_trans_out[i].trigger == e){
         defined_for = defined_for || state.all_trans_out[i].cond;
      }
   }
   for (j = i; j < size_of(state.all_trans_out); j++){
      if (state.all_trans_out[j].trigger == e){
         conflict_condition = state.all_trans_out[i].cond &&
                              state.all_trans_out[j].cond;
         if (conflict_condition != FALSE){
           create a new conflict;
           conflict.trans1 = state.all_trans_out[i];
           conflict.trans2 = state.all_trans_out[j];
           conflict.cond = conflict_condition;
           Append(conflict_array, conflict);
         }
      }
   }
   if(!defined_for) /* If there are missing conditions */
      output("No transition out of the state" state "is satisfied");
      output("under the event" e "if" !defined_for);
   output("There are conflicts between the following transitions:");
   output(conflict_array);
}
```

TABLE 4
DATA STRUCTURE FOR NONDETERMINISTIC PAIRS OF TRANSITION

```
struct Event{
   String          name;
   TransitionArray transitions; // All transitions triggered by this event
}
struct Nondeterministic{
   Trans trans1;
   Trans trans2;
}
```

TABLE 5
DETERMINISM UNDER EVENT e

```
NondeterministicArray NondeterministicTransitions(Event e)
{
  NondeterministicArray resultArray;
  for(i = 0 ; i < size_of(e.transitions); i++){
    for(j = i+1; j < size_of(e.transitions); j++){
      if (parallel((e.transition[i]),
                   (e.transition[j])) &&
          (conflicts(e.transitions[i].uses,
                     e.transitions[j].effects)||
           conflicts(e.transitions[j].uses,
                     e.transitions[i].effects))){
         AppendPair(result_array, e.transition[i],
                    e.transition[j]);
      }
    }
  }
  return resultArray;
}
```

A pairwise comparison of all parallel transitions can assure determinism: If no two transitions conflict, then the model is deterministic. Tables 4 and 5 outline the data structures and algorithm used for this analysis.

The pairwise comparison of all transitions existing in parallel and triggered by simultaneous events is potentially costly; in the worst case (all transitions are parallel), the algorithm requires $O(n^2)$ comparisons (where n is the number of transitions in the model). Fortunately, the number of parallel transitions in real systems seems to be fairly limited, and this straight forward approach has been shown to be adequate to analyze a major part of a large real life system (TCAS II) for determinism [13].

In summary, the algorithms described in this paper are all quite simple. This simplicity results from, and is an advantage of, our functional definition of the semantics of RSML. Unfortunately, the algorithms outlined above all have high worst-case complexity. For example, checking the union

compositions is exponential with respect to the size of the guarding conditions, and checking determinism is $O(n^2)$ with respect to the number of parallel transitions. However, by using our functional composition approach, all algorithms work on fairly small problems, i.e., individual compositions, and this complexity is acceptable. The compositional approach allows us to determine if these properties are maintained when hierarchies, parallelism, and event propagation are introduced and avoids the problems of combinatorial explosion of the problem size and exponential growth in analysis effort. Experiments (described in the next section) have shown that our approach to analysis can be effectively applied to large systems.

5 Automated Analysis Tools and Their Evaluation of the TCAS II System Requirements Specification

Manually assuring d-completeness and consistency is an extremely tedious, time-consuming, and error-prone task. Tool support for the analysis algorithms have been implemented as an integral part of a simulator for RSML. The simulator accepts a textual representation of RSML and allows execution of a requirements specification.

A prototype graphical interface allows browsing the specification and animating executions. The analysis tools outlined in the previous section are integrated into this simulator. In addition to the results from the analysis algorithms (reporting inconsistency, incompleteness, and nondeterminism), the tools generate other useful information, such as uses hierarchies and event propagation tables.

Although the TCAS specification effort was originally planned to be experimental only, the government/industry groups responsible for TCAS II liked RSML so much that the specification was adopted as the official FAA TCAS II System Requirements Specification [22]. As a result, our initial baseline specification was subjected to an extensive (and expensive) independent verification and validation (IV&V) effort.

We have applied the analysis techniques described in this paper to major parts of our baseline TCAS II specification. Initial comparison of the errors found during IV&V and by our automated analysis indicates that inconsistency problems found during IV&V were also found by our automated analysis tools. Some subtle inconsistency problems not found during the official IV&V process were also found.

The analysis procedures also found many instances of incompleteness. Unfortunately, we have not been able to correlate these results with the IV&V effort since the IV&V process did not include inspection for incompleteness. During IV&V, only the conditions under which state changes take place were reviewed; the conditions under which the state is not changed were not addressed.

The rest of this section provides some examples of the types of problems the analysis exposes. Drawbacks with the current implementation of the analysis procedures are also discussed.

5.1 D-Completeness

Because d-completeness was not a priority in our initial TCAS requirements development (highest priority was placed on simply getting what was specified correct), we found abundant incompleteness during the later analysis process. In retrospect, we believe that if we had had our completeness analysis tools to alert us to incompleteness as we were developing the specification, the resulting document would have been much more complete. An example from the baseline document suffices to illustrate both the complexity of developing d-complete requirements and some problems with the current implementation of the analysis tools.

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, i.e., an advisory will be generated earlier (while the planes are farther apart). This example is taken from Auto-SL, a concept of sensitivity level based mainly on the aircraft altitude. Consider the transition in Fig. 8. This transition defines when the model stays in Auto-SL state ASL-1. The automated analysis techniques detected an incompleteness—no transition out of the state is satisfied under a given condition (when a descend-inhibit-evaluated-event has occurred) shown in Fig. 9. The analysis result reflects all conditions under which no transition out of this state can be taken. The abundance of predicates results from the diversity of the guarding conditions on the other transitions out of this state. This diversity makes it extremely difficult to determine manually (without the assistance of our analysis tools) the conditions for which no behavior has been specified.

Transition(s): ASL-1 $\longrightarrow$ ASL-1

Location: Own-Aircraft ▷ Auto-SL$_{s\text{-}30}$

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

Condition:

			OR
AND	Own-Air-Status$_{v\text{-}36}$ = On-Ground	T	T
	Traffic-Display-Permitted$_{v\text{-}39}$	·	F
	Mode-Selector$_{v\text{-}34}$ = Standby	T	·

Output Action: Auto-SL-Evaluated-Event$_{e\text{-}279}$

Fig. 8. The identity transition for Auto-SL state ASL-1.

Given the output shown in Fig. 9, the analyst can determine what response the model should have for all conditions identified by the tool and modify the guarding conditions on the transitions to make the model d-complete (Fig. 10). In this case the desired behavior was to stay in ASL-1 under all conditions identified by the analysis. In the general case, it is likely that more than one transition will need to be modified in order to cover these "forgotten" conditions. With this modification, the set of transitions out of state ASL-1 is d-complete, and the tool will report that there are no conditions where the behavior is unspecified (Fig. 11).

```
No transition out of ASL_1
is satisfied under Descend_Inhibit_Evaluated_Event if :

Own_Air_Status == OnGround                 : F F F F F F F F F F F
Own_Air_Status == Airborne                 : T T T T T T T T T T T
Traffic_Display_Permitted == cTrue         : F F F F T T T T T T F
Effective_SL In One Of {ESL1,ESL2,ESL3}    : F F F F F F F F F F F
Effective_SL In State ESL4                 : T F F F T T T F F F T
Effective_SL In State ESL5                 : F T F F F F F T F F F
Effective_SL In State ESL6                 : F F T F F F F F T F F
Effective_SL In State ESL7                 : F F F T F F F F F T F
Own_Alt_Barometric >= ZSL4TO5              : . . . . . . . . . . .
Own_Alt_Barometric >= ZSL5TO6              : . F . . . . . F . . .
Own_Alt_Barometric >= ZSL6TO7              : . . F . . . . . F . .
Own_Alt_Barometric <= ZSL6TO5              : . . F . . . . . F . .
Own_Alt_Barometric <= ZSL7TO6              : . . . F . . . . . F .
Own_Alt_Radio <= ZSL4TO2                   : F . . . T F F . . . T
Own_Alt_Radio >= ZSL4TO5                   : . . . . . . F . . . .
Own_Alt_Radio > ZSL5TO4                    : . T . . . . . T . . .
Own_Alt_Radio <= ZSL5TO4                   : . F . . . . . F . . .
```

Fig. 9. D-completeness analysis result for Auto-SL state ASL-1.

Transition(s): ASL-1 ⟶ ASL-1

Location: Own-Aircraft ▷ Auto-SL$_{s\text{-}30}$

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

Condition:

OR

	Own-Air-Status$_{v\text{-}36}$ = On-Ground	T	T	F	F	F	F	F	F	F	F	F	F	F	F
	Traffic-Display-Permitted$_{v\text{-}39}$	·	F	·	·	·	·	·	·	·	·	·	·	·	·
	Mode-Selector$_{v\text{-}34}$ = Standby	T	·	·	·	·	·	·	·	·	·	·	·	·	·
	Climb-Desc-Inhibit()	·	·	F	F	F	F	F	F	F	F	F	F	F	F
	Own-Air-Status$_{v\text{-}36}$ = Airborne	F	F	T	T	T	T	T	T	T	T	T	T	T	T
	Radar-Bad-For-RADARLOST-Cycles()	·	·	F	·	·	·	·	·	·	·	F	·	·	·
	Radarout-EQ-0()	·	·	F	·	·	F	·	·	F	F	F	F	F	T
	Effective-SL **in one of** ESL-1,ESL-2,ESL-3	·	·	T	·	·	·	·	·	·	·	·	·	·	·
A	Effective-SL **in state** ESL-4	·	·	·	·	·	·	·	T	T	T	T	T	T	T
N	Effective-SL **in state** ESL-5	·	·	·	·	·	T	T	·	·	·	·	·	·	·
D	Effective-SL **in state** ESL-6	·	·	·	·	T	·	·	·	·	·	·	·	·	·
	Effective-SL **in state** ESL-7	·	·	·	T	·	·	·	·	·	·	·	·	·	·
	Own-Alt-Barometric$_{v\text{-}33}$ ≥ ZSL4TO5	·	·	·	·	·	·	·	F	F	T	F	F	T	T
	Own-Alt-Barometric$_{v\text{-}33}$ ≥ ZSL5TO6	·	·	·	·	·	F	F	·	·	·	·	·	·	·
	Own-Alt-Barometric$_{v\text{-}33}$ ≥ ZSL6TO7	·	·	·	·	F	·	·	·	·	·	·	·	·	·
	Own-Alt-Barometric$_{v\text{-}33}$ ≤ ZSL6TO5	·	·	·	·	F	·	·	·	·	·	·	·	·	·
	Own-Alt-Barometric$_{v\text{-}33}$ ≤ ZSL7TO6	·	·	·	F	·	·	·	·	·	·	·	·	·	·
	Own-Alt-Radio$_{v\text{-}31}$ ≤ ZSL4TO2	·	·	·	·	·	·	·	F	T	·	F	F	F	
	Own-Alt-Radio$_{v\text{-}31}$ ≤ ZSL5TO4	·	·	·	·	·	T	F	F	·	·	T	F	F	
	Own-Alt-Radio$_{v\text{-}31}$ ≥ ZSL4TO5	·	·	·	·	·	·	·	F	·	·	·	T	F	

Output Action: Auto-SL-Evaluated-Event$_{e\text{-}279}$

Fig. 10. Transition modified for d-completeness.

```
No transition out of ASL_1
is satisfied under Descend_Inhibit_Evaluated_Event if :

FALSE
```

Fig. 11. Analysis result for the modified specification.

5.1.1 Spurious Error Reports

During initial experiments with our first prototype tool, spurious error reports were not a serious problem [14]. All spurious reports could be traced either to 1) a lack of type checking capability 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 input variable Air-Status of the enumerated type {Airborne, On-Ground} (appearing in the first two rows of Fig. 9). Without information about the all inclusive and mutually exclusive nature of enumerated types, the tool would generate additional error reports and indicate that additional transitions out of ASL-1 are needed for the case

A		
N	Air-Status = Airborne	F
D	Air-Status = On-Ground	F

This is a clearly erroneous report because the input Air-Status must have one of these values; we do not need to specify what to do under this unsatisfiable condition. Similar problems relating to the structure of the state machine also led to spurious error reports. These drawbacks were trivial to address, and an updated version of the tool eliminates our previous problems with spurious errors.

Unfortunately, these changes do not eliminate all spurious error reports. Two features of the predicates in RSML complicates the analysis: 1) the use of simple arithmetic and 2) the use of mathematical functions. Contradictory predicates involving these features cannot be detected by the symbolic BDD approach. The number of spurious error reports increases dramatically when the number of predicates including these features increases. For example, the analysis tool may generate an error report including the condition in Table 6 (indicating that no transition has been specified for this condition). Any error report containing this condition is spurious because the predicates in Table 6 cannot be satisfied simultaneously. The current implementation of our tool is unable to eliminate this type of spurious error report. The problem is amplified when predicates use references to mathematical functions instead of constant values.

TABLE 6
A SPURIOUS REPORT OF AN OMITTED CONDITION

A	Other-Tracked-Range-Rate$_{f\text{-}245}$ > 10	T
N	Other-Tracked-Range$_{f\text{-}245}$ > 0.55	T
D	|Other-Tracked-Range-Rate$_{f\text{-}245}$ · Other-Tracked Range$_{f\text{-}245}$| ≤ 0.00278	T

The problem with simple arithmetic expressions in the predicates can be addressed by using a theorem prover. Currently, however, the conflicts must be detected and eliminated by manual inspection. An ongoing project is attempting to augment our tool with theorem proving capability, and we hope to eliminate the problems with arithmetic expressions shortly.

The use of references to named mathematical functions in the definition of guarding conditions is a more serious challenge. We are investigating how assertions or invariants associated with the functions can be used to further increase accuracy. Unfortunately, completely eliminating spurious errors while still maintaining reasonable efficiency is an unrealistic goal. Thus, tool support to help the human analyst to interpret the analysis results and detect such problems manually is also being developed.

5.2 Consistency

A consistency problem exists when the guarding condition on more than one transition can be satisfied simultaneously.

The state machine modeling Effective-SL (which is related to Auto-SL) is shown in Fig. 12. The bar on the side is a transition bus. Many state machines in the model were found to be fully interconnected, i.e., there are 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. 13). The inconsistency (as reported from the analysis tool) can be seen in Fig. 14: Column 3 of both transitions are satisfied by the condition. Since sensitivity level ESL-5 represents a sensitive setting and ESL-2 represents that advisories are shut 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 (Fig. 15).

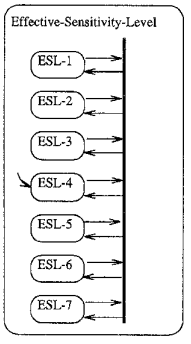

Fig. 12. Effective sensitivity level.

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		
A	Auto-SL$_{s\text{-}30}$ **in state** ASL-5	T	T	T	·	·
N	Auto-SL$_{s\text{-}30}$ **in one of** {ASL-5,ASL-6,ASL-7}	·	·	·	T	T
D	Lowest-Ground$_{f\text{-}241}$ **= one of** {5,6,7,None}	T	·	·	·	T
	Lowest-Ground$_{f\text{-}241}$ = 2	·	·	T	·	·
	Lowest-Ground$_{f\text{-}241}$ = 5	·	·	·	T	·
	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. 13. The transition from Effective-SL ESL-4 to ESL-5.

```
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 Equals One Of {TA_RA,TA_Only,3,4,5,6,7} : T
Auto_SL In State ASL_5                          : T
```

Fig. 14. 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	Auto-SL$_{s\text{-}30}$ **in state** ASL-2	T
N	Mode-Selector$_{v\text{-}34}$ = Standby	F
D		

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

Fig. 15. The modified transition from Effective-SL ESL-4 to ESL-2.

Unfortunately, correcting an inconsistency is often not as simple as strengthening the guarding condition on one of the transitions involved in the inconsistency. Inconsistencies sometimes arose from logical errors in the requirements and an extensive redesign of that part of the requirements document was needed.

Other approaches to requirements specification analysis are not concerned with this kind of inconsistency—it is simply viewed as nondeterminism and accepted as a part of the requirements. As was mentioned in Section 3, we view nondeterminism as an inconsistency that should, in most cases, be eliminated. At the least, each case needs to be carefully examined because nondeterminism can have a negative effect on safety (as shown by the example in this section).

6 Conclusions

This paper outlines a functional framework enabling compositional static analysis of state-based requirements and shows how the analysis for two fundamental qualities of requirements specifications—d-completeness and consistency—can be automated. The feasibility of the analysis has been demonstrated by analyzing major parts of a real life avionics system (TCAS II). The approach outlined in this paper has several advantages:

- The analysis does not require generation of any part of the global reachability graph (either a complete representation or a symbolic representation).
- It enables incremental analysis of the requirements. The pieces of the requirements document can be analyzed as they are being developed and the individual results combined at a later stage.
- It helps identify the parts of the requirements needing reanalysis after changes to the document have been made.
- It is a conservative approach, i.e., it is guaranteed that no d-incompleteness, inconsistency, or nondeterminism will go undetected.

We get these advantages by limiting the semantics of the specification language to those that can be described by functional composition. In doing this, we give up some freedom both in defining the semantics of the language and in the models that we allow users to build. We believe, however, that the increased power of the analysis that we can perform on complex models in comparison to other current approaches makes the tradeoff worthwhile. We found that eliminating the nondeterminism from the language made it easier for the TCAS reviewers to understand the model and find errors in it. So our restrictions have advantages in reviewability, correctness, and analysis, but they do cause some loss of flexibility in language design.

Because the BDDs we use to represent our AND/OR tables manipulate predicates symbolically, the analysis is conservative and may generate spurious error reports. The main source of spurious reports is the use of arithmetic and function references in the predicate definitions. Our tool is currently being refined to correct this problem. We are investigating the tradeoffs between efficiency and accuracy, and we are integrating the symbolic BDD approach with a theorem prover to achieve the level of accuracy required to easily interpret analysis results from the most complex parts of the TCAS requirements.

Our long term goal is to provide a suite of analysis tools to help find a wide variety of flaws in software requirements early during software development. Many desirable properties of requirements specification have been defined by Jaffe et al. [20], for example, nonreachability of hazardous states and path robustness properties. Additional properties are being defined for the human–computer interface (see Leveson [24] for some of the new criteria). Our goal is to formally define these properties in the RSML framework (and develop new ones suitable to this new framework) and provide efficient automated analysis procedures for these properties.

Appendix A – Auxiliary Definitions

The definitions in this appendix describe the hierarchical and parallel structure of the state machine used in both RSML and Statecharts. The definitions are adopted from [12], [26].

Definition 11. *The least common parent (lcp) of the states in the set $X \in 2^S$, $lcp(X) = y$, is defined as the supremum of the elements in X.*

The equivalence relation $\sim$ divides the descendants of any given state into parallel components.

Definition 12. *States a and b are parallel substates of x ($a \perp b$) iff:*

$$\exists\, u, v \in \sigma(x) \;:\; \neg\,(u \sim v) \wedge (a \leq u) \wedge (b \leq v)$$

Informally, the states a and b are parallel iff they are descendants (according to $\leq$) of inequivalent (under $\sim$) children of x. In Fig. 6, for example, H and B, B and D, and F and H are all pairwise parallel. Examples of nonparallel states include 1) B and C and 2) D and H.

Definition 13. *A set $X \in 2^S$ is said to be parallel iff:*

$$\forall x, y \in X : (x = y) \vee (x \perp y)$$

That is, all elements of X are pairwise parallel. The set $\{B, G, I\}$ is an example of a parallel set.

Definition 14. *A set $X \in 2^P$ is consistent iff:*

$$\forall x, y \in X : (x \leq y) \vee (y \leq x) \vee (x \perp y)$$

Informally, a set of states x is consistent iff all states in X are either ancestrally related or parallel. As an example, for the states in Fig. 6, the sets $\{F, H\}$ and $\{E, H\}$ are both consistent, but $\{D, H\}$ is not.

Definition 15. *A set $X \in 2^S$ is said to be maximally consistent iff:*

$$\forall\, x \in S - X : \neg \mathit{consistent}(X \cup \{x\})$$

The concept of maximally consistent is best explained with an example. Consider Fig. 6 and assume the machine is in state H. Given the structure of the state hierarchy, it is clear that if you are in H, you also are in E and A. The concept of maximally consistent "fills in the blanks" in a consistent set: If you are in H, you also *have* to be in E and A. A maximally consistent set of states is known as a *configuration* of M and *Config* is defined as the set of all configurations. The set $\{A, B, E, F, H\}$ is maximally consistent. This set defines one possible configuration of M.

We can now formally define the set of *global states* of M:

DEFINITION 16. *The set of global states C is defined as:*

$$C = \{c \mid c \subseteq 2^S \wedge \textit{max-consistent}(c)\} \times V$$

This concludes the formal definition of the structure of the states making up the graphical notation used in RSML (and Statecharts). Note again that these definitions are essentially identical to the definition of the hierarchical structure of Statecharts [12], [26].

ACKNOWLEDGMENTS

This work has been partially supported by the National Science Foundation Grant CCR-9006279, NASA Grant NAG-1-668, and the National Science Foundation CER Grant DCR-8521398.

We would like to thank David Guaspari from Odyssey Research Associates for his feedback on earlier drafts of this paper and his invaluable help with the formal definition of the RSML semantics. David Guaspari was partially supported by the Office of Naval Research, contract number N0014-95-5-0349.

REFERENCES

[1] J. Atlee and J. Gannon, "State-Based Model Checking of Event-Driven System Requirements," *Proc. ACM SIGSOFT '91 Conf. Software for Critical Systems. Software Engineering Notes*, vol. 16, No. 5, 1991.

[2] G.R. Bruns, S.L. Gerhart, I. Forman, and M. Graf, "Design Technology Assessment: The Statecharts Approach," Technical Report STP-107-86, MCC, Mar. 1986.

[3] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[4] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," Technical Report CMU-CS-93-211, Carnegie Mellon Univ., July 1993.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Proc. Fifth Ann. Symp. on Logic in Computer Science*, June 1990.

[6] E.M. Clarke, M.C. Browne, E.A. Emerson, and A.P. Sistla, "Using Temporal Logic for Automatic Verification of Finite State Systems," K.R. Apt, ed., *Logics and Models of Concurrent Systems*, pp. 3–26. Berlin: Springer-Verlag, 1985.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[8] P. Godefroid, G.J. Holzmann, and D. Pirottin, "State Space Caching Revisited," *Proc. Fourth Workshop Computer-Aided Verification*, pp. 175–186, 1992.

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

[10] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," Technical Report CS95-31, The Weizmann Institute of Science, Oct. 1995.

[11] D. Harel and A. Pnueli, "On the Development of Reactive Systems," K.R. Apt, ed., *Logics and Models of Concurrent Systems*, pp. 477–498. Springer-Verlag, 1985.

[12] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman, "On the Formal Semantics of Statecharts (extended abstract)," *Proc. Second Symp. Logic in Computer Science*, pp. 54–64, Ithaca, N.Y., 1987.

[13] M.P.E. Heimdahl, "Static Analysis of State-Based Requirements: Analysis for Completeness and Consistency," PhD thesis, Univ. of California, Irvine, 1994.

[14] M.P.E. Heimdahl and N.G. Leveson, "Completeness and Consistency Analysis of State-Based Requirements, *Proc. 17th Int'l Conf. Software Engineering*, Apr. 1995.

[15] C.L. Heitmeyer, B.L. Labaw, and D. Kiskis, "Consistency Checking of SCR-style Requirements Specifications," *Proc. Int'l Symp. Requirements Engineering*, Mar. 1995.

[16] K.L. Heninger, "Specifying Software for Complex Systems: New Techniques and their Application," *IEEE Trans. Software Engineering*, vol. 6, no. 1, pp. 2–13, Jan. 1980.

[17] K.L. Heninger, J.W. Kallander, J.E. Shore, and D.L. Parnas, Software Requirements for the A-7e Aircraft. Technical Report 3876, Naval Research Laboratory, Washington, D.C., Nov. 1978.

[18] G.J. Holzmann. "Automated Protocol Validation in Argos: Assertion Proving and Scatter Searching," *IEEE Trans. Software Engineering*, vol. 13, no. 6, pp. 683–696, June 1987.

[19] G.J. Holzmann, "Tracing Protocols," *AT&T Technical J.*, vol. 64, no. 10, Dec. 1985.

[20] M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl, and B. Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Trans. Software Engineering*, vol. 17, no. 3, pp. 241–258, Mar. 1991.

[21] M.S. Jaffe, "Completeness, Robustness, and Safety in Real-Time Software Requirements and Specifications," PhD thesis, Univ. of California, Irvine, 1988.

[22] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J. Reese, "TCAS II Requirements Specification,"

[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*, vol. 20, no. 9, Sept. 1994.

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

[25] R. Lutz, "Targeting Safety-Related Errors During Software Requirements Analysis," *Proc. First ACM SIGSOFT Symp.The Foundations of Software Engineering*, 1993.

[26] A. Pnueli and M. Shalev, "What is in a Step?" J. Klop, J. Meijer, and J. Rutten, eds., *J.W. De Baker, Liber Amicorum*, pp. 373–400. CWI Amsterdam, 1989.

[27] A.P. Ravn and H. Richel, "Requirements Capture for Embedded Real-Time Systems," *IMACS Symp. MCTS*, 1991.

[28] H. Richel and A.P. Ravn, "Requirements Capture for Computer Based Systems," Technical Report ID/DTH HR 2/2, Technical Univ. of Denmark, Oct. 1990.

[29] J. Rushby and F. von Henke, "Formal Verification of Algorithms for Critical Systems," *IEEE Trans. Software Engineering*, vol. 19, no. 1, pp. 13–23, Jan. 1993.

Mats P.E. Heimdahl received the MS degree in computer science and engineering from the Royal Institute of Technology, Stockholm, Sweden in 1988, and the PhD degree in information and computer science from the University of California at Irvine in 1994. He is currently an assistant professor in the computer science department at the University of Minnesota, Twin Cities. Currently, his research interests are in requirements specification, static analysis of state-based models, software development for critical systems, executable specification, and formal methods.

Nancy G. Leveson is Boeing professor of computer science and engineering at the University of Washington. She received a bachelor's degree in Math, an MS degree in management (operations research), and a PhD degree in computer science from UCLA. Her research interests are in software safety and reliability. She is the author of the book *Safeware: System Safety and Computers*. Dr. Leveson is an elected member of the board of directors of both the Computing Research Association and the International Council on System Engineering. She is member of the National Research Council (National Academy of Science) Commission on Engineering and Technical Systems, a liaison to the NRC Aeronautics and Space Engineering Board, and a member of the ACM Committee on Computers and Public Policy. She was named a fellow of the ACM and is the 1995 recipient of the AIAA Information Systems Award for "developing the field of software safety and for promoting responsible software and system engineering practices where life and property are at stake." Dr. Leveson chaired the National Academy of Science's study committee for Review of Oversight Mechanisms for Space Shuttle Flight Software Processes, was the U.S. representative to the International Atomic Energy Agency Committee on Software in Nuclear Power Plants, and is currently a member of an NRC study committee on Digital Instrumentation and Control in Nuclear Power Plants. Dr. Leveson consults widely on safety-critical systems for both government and industry.