Requirements Specification for Process-Control Systems *†

Nancy G. Leveson
Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Mats P.E. Heimdahl
Computer Science Department, A-714 Wells Hall
Michigan State University
East Lansing, Michigan 48824-1027

Holly Hildreth
Jon D. Reese
Information and Computer Science Dept.
University of California, Irvine
Irvine, CA 92717

*This work has been partially supported by NSF Grant CCR-9006279, NASA Grant NAG-1-668, and NSF CER Grant DCR-8521398.

†This paper has appeared in *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–107, September 1994.

Abstract

This paper describes an approach to writing requirements specifications for process-control systems, a specification language that supports this approach, and an example application of the approach and the language on an industrial aircraft collision avoidance system (TCAS II). The example specification demonstrates (1) the practicality of writing a formal requirements specification for a complex, process-control system and (2) the feasibility of building a formal model of a system using a specification language that is readable and reviewable by applications experts who are not computer scientists or mathematicians. Some lessons learned in the process of this work, which are applicable both to forward and reverse engineering, are also presented.

Index Terms: process control, reactive systems, requirements, blackbox specifications, formal methods, safety analysis, reverse engineering

1 Introduction

Embedded software is part of a larger system and has a primary purpose of providing at least partial control of the system or process in which it is embedded. Most such software is real-time and reactive (i.e., required to interact with and respond to its environment in a timely fashion during execution). A high cost is associated with determining the correctness of such software and a still higher cost associated with its incorrectness. The requirements for complex, embedded software systems are particularly difficult to specify and validate.

The very first stages of software development have the fewest formal procedures to aid the analyst, and this is also the time at which the most costly errors are introduced in terms of being the last and most difficult to find. Many software requirements validation techniques involve building prototypes or executable specifications or waiting until the software is constructed and then testing the whole system. Although certainly much can be learned by "testing" a specification through executing it, or a prototype built from it, the confidence that the system will have certain properties is limited to the test cases that were executed. Our approach is to model the required software blackbox behavior along with the assumptions about the behavior of the other components of the system, and then to apply formal analysis procedures to the model in order to ensure that the software requirements model satisfies required system functional goals and constraints, including safety.

Several different safety analysis procedures have been developed by members of the Irvine Safety Research Group [LH83, LS87, LCS91, JLHM91], but they work on diverse models and have not been validated on real software. Our long-term goal is to develop a coherent, complete, and practical methodology for building safety-critical systems. This paper concentrates on the earliest part of the methodology, i.e., requirements specification, and demonstrates it on a real system. Future papers will describe the analysis procedures we are developing and evaluating for our model.

Most of the information to be included in our system requirements model already is collected by system engineers or software engineers. However, the information is commonly scattered throughout the system documentation, is usually informally specified, and is not in a form amenable to formal analysis. In addition, the information is often specified using multiple different and incompatible models within the same specification (e.g., Statemate [HLN+90], Hatley/Pirbhai [HP87], Ward/Mellor [WM85]). For example, Statemate uses Statecharts, Activity Charts, and Structure Charts; Hatley/Pirbhai uses data flow diagrams, control flow diagrams, control specifications (finite state machines), and a process activation table.

Our approach is to build one state-based model that includes all of the information needed to describe the black-box behavior of the components of the system (including but not only the computer) and the interface between the components and no more. By having all the requirements information in one model, formal analysis of the entire system becomes feasible and redundancy is reduced. The latter reduces the difficulty of changing the specification without introducing inconsistency.

Furthermore, a blackbox model separates the specification of requirements from design, simplifying the model and making the requirements model easier to construct, review, and formally analyze. Most software requirements specification languages include software design information; the original A-7 specification [Hen80] is a notable exception. The modeling language described in this paper differs from the A-7 language, however, in the use of higher-level, global abstractions of the entire system and in the goal of providing formal system analysis procedures to operate on the underlying formal model. Research has increased on the development of higher-level abstractions for embedded systems [FBWJ92, vS90]; and although large-scale examples are still lacking, some of these more recent ideas are being applied in retrospect to the A-7 system.

Finally, the language defined here has analysis goals similar to the ProCoS system [RR91], but uses state machines whereas ProCoS uses process algebras. Consequently, the analysis procedures applicable to our model are related to ongoing work in automated state space analysis [Hol91, CES86] while the ProCoS approach relies on traditional methods of theorem proving to analyze their models. Our approach is also similar to some recent work by Parnas [PW89], which also uses tables and state machines but uses trace semantics for analysis.

The most important result of our research is verification that building a formal requirements model for a complex process control system is possible and that such a model can be readable and reviewable by non-computer scientists. Few examples exist of the application of formal methods to a complex, reactive system requirements specification. In order to evaluate our safety analysis ideas, we needed to build a model of a realistic system to use as a test bed. This paper describes the resulting formal system modeling method and its use to specify the system requirements of an aircraft collision avoidance system called TCAS II. In the midst of this effort, our model was adopted as the official requirements specification for TCAS II, so the research effort and the resulting model had to be industrial quality. The unique part of this effort, at least in terms of university research, is that the specification language was developed with continual feedback and evaluation by FAA employees, airframe manufacturers, TCAS manufacturers, airline representatives, pilots, and other external reviewers. Most of the reviewers were not software engineers or even computer scientists; this helped in producing a specification language that is easily learned and used by application experts.

Although we describe a particular language that we used for this model, the details of the actual language features are less important than the other results of the research: (1) the general criteria that any such modeling method and language must satisfy, (2) the type of information that must be included in such a system requirements model in order for it to be analyzable for safety, and (3) the required features of such a language in order to make it possible to model real systems and to be usable by application experts. All of these are described in this paper. Future papers will describe the actual application of safety analysis techniques to the model.

Our results have both forward engineering and reverse engineering implications. A detailed design specification written in low-level pseudocode (about 300 pages long) already existed for most of our application. Other parts, however, had only an English language description. Some of the lessons learned about reverse engineering are described in this paper.

The next section briefly describes the application, a collision avoidance system called TCAS II. This is followed by an overview of the specification approach and descriptions of both the language and the system requirements specification.

2 The Traffic Alert and Collision Avoidance System (TCAS)

A real aircraft collision avoidance system (called TCAS II) was used as a testbed to provide immediate evaluation and feedback for our modeling and analysis ideas. TCAS II has been described by the head of the 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.

TCAS is a family of airborne devices that function independently of the ground-based air traffic control (ATC) system to provide collision avoidance protection for a broad spectrum of aircraft types (commercial aircraft and larger commuter and business aircraft). TCAS I provides proximity warning (traffic advisories) to assist the pilot in the visual sighting of intruder aircraft and is intended for use by smaller commuter and general aviation aircraft. TCAS II provides traffic advisories and recommended escape maneuvers (resolution advisories) in a vertical direction to avoid conflicting aircraft. TCAS III will add resolution advisories in a horizontal direction.

Development of aircraft collision avoidance systems started over 20 years ago. In 1981, the FAA decided to develop and implement TCAS II, and a Minimal Operational Performance Standards (MOPS) document was produced using a combination of English and pseudocode. Since its adoption in 1983, the MOPS has been extensively revised six times to fix errors or improve the specification. In 1989, the FAA required that TCAS II be installed on commercial aircraft with more than 30 seats by December 1991 and on commercial aircraft with 10 to 30 seats by 1995. The FAA relaxed the first deadline to require installation on half the commercial aircraft fleet by 1991 and on the remainder by 1993.

The MOPS document contains information that we would classify as system design (in English) and software design (in English and pseudocode). Because of perceived deficiencies in this document and the difficulty of FAA certification without real system or software requirements, an effort was begun in 1990 to provide a requirements document for TCAS II. An industry/government committee began to write a fairly standard English language specification while we started an experimental formal specification and safety analysis. Our specification was subsequently adopted by the committee as the official TCAS requirements specification and the other specification effort was abandoned.

3 Specifying Requirements for Process-Control Systems

3.1 Goals, Constraints, and Requirements

A system is a set of components working together to achieve some common purpose or objective. The requirements specification language being described in this paper was designed for process control systems, where the goal is to maintain a particular relationship or function F over time (t) between the input to the system ($\mathcal{I}_s$) and the output from the system ($\mathcal{O}_s$) in the face of disturbances ($\mathcal{D}$) in the process (see Figure 1). These relationships will involve fundamental chemical, thermal, mechanical, aerodynamic or other laws as embodied within the nature and construction of the system.

Besides the basic objective or function implemented by the process, these types of systems may also have constraints on their operating conditions. Constraints may be regarded as boundaries that define the range of conditions within which the system may operate. Another way of thinking about constraints is that they limit the set of acceptable designs with which the objectives may be achieved.

Constraints may arise from several sources, including quality considerations, physical limitations and equipment capacities (e.g., avoiding equipment overload in order to reduce maintenance), process characteristics (e.g., limiting process variables to minimize production of byproducts), and safety (i.e., avoiding hazardous states). In some systems, the functional goal is to maintain safety, so safety is part of the overall objective as well as potentially part of the constraints.

As an example, for an airborne collision avoidance system like TCAS, $\mathcal{I}_s$ can be viewed as all aircraft that fly into the airspace of the TCAS-equipped aircraft and $\mathcal{O}_s$ as all aircraft that fly out of the airspace of the TCAS aircraft. The goal of the TCAS system is to maintain a minimum separation function between the aircraft. Constraints include such things as not interfering with the ground-based air traffic control (ATC) system, operating with an acceptably low level of unwanted alarms (advisories to the pilot), and minimizing the amount of deviation of the aircraft from their ATC-assigned tracks.

Note that the goals of a system are just that, i.e., they may not be entirely achievable. Although the goal of TCAS II is to *eliminate* near-misses (i.e., aircraft violating minimum separation standards), this cannot be a requirement since it is not possible to achieve: It is, however, a legitimate goal. Another way of stating this goal is to *minimize* the number of near-misses. The latter, however, is not a measurable goal since its achievement cannot be determined. Another possibility that theoretically can be evaluated is to *reduce* near-misses. The amount of reduction that is actually achieved then becomes a criterion for whether the system can be justified based on cost and possible increased risk with respect to other hazards in the system. The point here is that goals are different from requirements because the goals may not be achievable. The actual required and achieved behavior can be evaluated with respect to the goals and constraints to determine whether the system, as specified and designed, is *acceptable*.

Early in the development process, tradeoffs between functional goals and constraints that are conflicting or not completely achievable must be identified and resolved according to the priorities assigned to them. Identifying these conflicts and resolving them is a major task in both the system and software requirements analysis process. A second task is ensuring that the specified (or required) behavior of the process-control system will achieve the goals to an acceptable degree while satisfying the constraints. Semantic analysis of our system requirements model can potentially address both of these elements of correctness since it includes a model of the behavior of all the components of the system.

3.2 Purpose and Content of Requirements Specification for Process-Control Systems

A typical process-control system can be divided into four types of components: the process, sensors, actuators, and controller (see Figure 1).

The behavior of the *process* is monitored through *controlled variables* ($\mathcal{V}_c$) and controlled by *manipulated variables* ($\mathcal{V}_m$). The process can be described by the process function F_P, a mapping from $\mathcal{V}_m \times \mathcal{I}_s \times \mathcal{D} \times t \rightarrow \mathcal{O}_s \times \mathcal{V}_c$. Unfortunately, it is usually difficult to

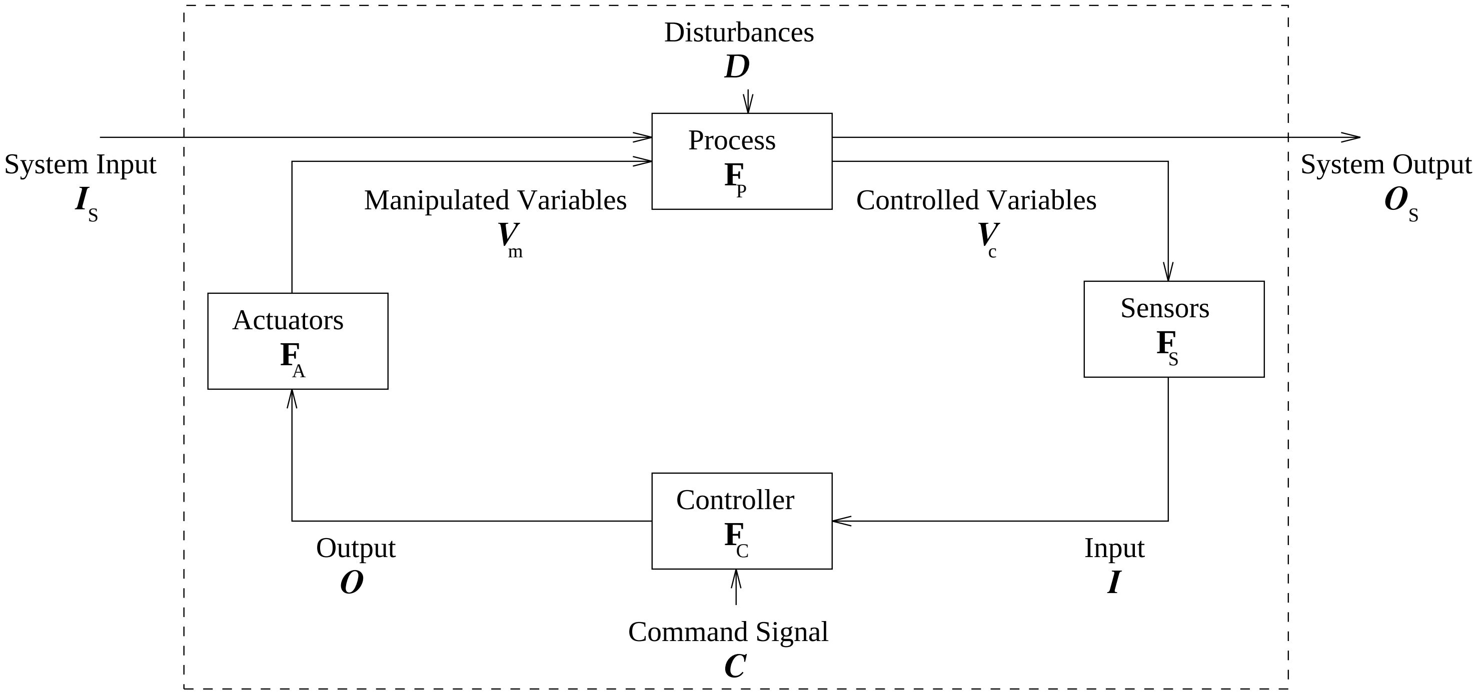


Figure 1: A basic process control model

derive a mathematical model of the process due to the fact that most processes are highly nonlinear (i.e., the process characteristics depend on the level of operation), and, even at a constant operating level, the process characteristics change with time (i.e., the process is nonstationary). Any attempt to provide a mathematical expression describing the process involves simplifying assumptions and therefore will be imperfect. Some of the process characteristics, however, can be described, and this description can be used to derive and validate the control function.

Sensors are used to monitor the actual behavior of the process by measuring the controlled variables. For example, a thermometer may measure the temperature of a solvent in a chemical process or a barometric altimeter may measure altitude of an aircraft above sea level. The sensor function F_S maps $\mathcal{V}_c \times t \rightarrow \mathcal{I}$.

Actuators are devices designed to manipulate the behavior of the process, e.g. valves controlling the flow of a fluid or a pilot changing the direction and speed of an aircraft. The actuators physically execute commands issued by the controller in order to change the manipulated variables. The functionality of the actuators is described by the actuator function F_A mapping $\mathcal{O} \times t \rightarrow \mathcal{V}_m$.

The *controller* is an analog or digital device used to implement the control function. The functional behavior of the controller is described by a control function (F_C) mapping $\mathcal{I} \times \mathcal{C} \times t \rightarrow \mathcal{O}$, where $\mathcal{C}$ denotes external command signals. The process may change state not only through internal conditions and through the manipulated variables, but also by disturbances ($\mathcal{D}$) that are not subject to adjustment and control by the controller. The general control problem is to adjust the manipulated variables so as to achieve the system goals despite disturbances.

This model is an abstraction—responsibility for implementing the control function may actually be distributed among several components including analog devices, digital computers, and humans. Furthermore, the controller may have only partial control over the process—state changes in the process may occur due to internal conditions in the process or because of external disturbances or the actuators may not perform as expected. For example, the pilot in a TCAS system may not follow the resolution advisory (escape maneuver) issued by the TCAS controller.

The purpose of the control-system requirements specification is to define the system goals and constraints, the function F_C (i.e., the required blackbox behavior of the controller), and the assumptions about the other components of the process-control loop that (1) the implementors need to know in order to implement the control function correctly and (2) the system engineers and analysts need to know in order to validate the model against the system goals and constraints.

A blackbox, behavioral specification of the function F_C uses only:

(1) the current process state inferred from measurements of the controlled variables,
(2) past process states that were measured and inferred,
(3) past corrective actions output from the controller, and
(4) prediction of future states of the controlled process

to generate the corrective actions (or current outputs) needed to maintain F.

Information about the process state has to be inferred from measurements. For example, in TCAS, relative range positions of other aircraft are computed based on round-trip message propagation time. Theoretically, the function F_C can be defined using only the true values of the controlled variables or component states (e.g., true aircraft positions). However, at any time, the controller has only measured values of the component states (which may be subject to time lags[1] or measurement inaccuracies), and the controller must use these measured values to infer the true conditions in the process and possibly to output corrective actions ($\mathcal{O}$) to maintain F. In the TCAS example, sensors include on-board devices such as altimeters that provide measured altitude (not necessarily true altitude) and antennas for communicating with other aircraft. The primary TCAS actuator is the pilot, who may or may not respond to system advisories. Pilot response delays are important time lags that must be considered in designing the control function. Time lags in the actual process may be caused by aircraft performance limitations.

[1]Time lags are delays in the system caused by the reaction time of the sensors, actuators, and the actual process.

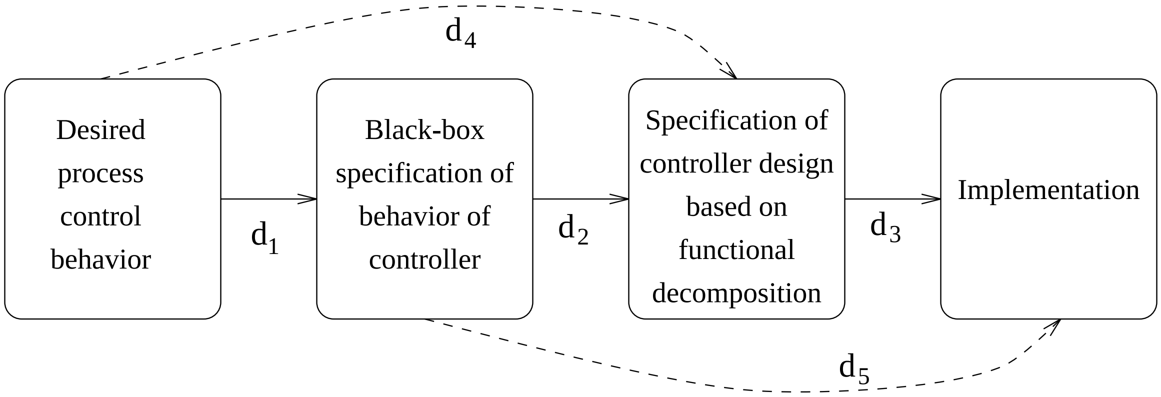


Figure 2: Semantic distance

3.3 An Approach to Writing Requirements for Process-Control Systems

We specify the blackbox behavior of the controller (i.e., the function F_C to be computed by the controller) using a state machine model. The outputs of the controller are specified with respect to state changes in the model as information is received about the current state of the controlled process via the controlled variables $\mathcal{V}_c$. In the TCAS example, the control function is specified using a model of the state of all other aircraft within the host aircraft's airspace, the state of the on-board components of its own aircraft (e.g., altimeters, aircraft discretes[2], cockpit display), and the state of ground-based radar stations in the vicinity. Information about this state is received from the sensors (e.g., antennas and transponders) and commands are sent to the actuators (e.g., the pilot, antennas, and transponders).

The state machine model of the control function F_C is iteratively fine tuned during requirements specification development to mimic the current understanding of the real-world process and the required controller behavior. The state machine model is essentially an abstraction of the behavior of the system function F since it models all the relevant aspects of the components of the process control loop. Errors in the state machine model represent mismatches between this model and the desired behavior of the control loop, including the process. We define the informal concept of *semantic distance* as the amount of effort required to translate from one model to another. We believe that in order to maximize the application expert's ability to find errors in the requirements specification, the semantic distance (d_1 in Figure 2) between their understanding of the desired process control behavior (their mental model of the system) and the specification of that behavior must be minimized. This, in turn, implies that the requirements be written entirely in terms of the

[2]Aircraft Discretes are airframe-specific characteristics provided as input to TCAS from hardware switches.

components and state variables of the controlled system. Specifically, "private" variables related only to the implementation of the requirements and not part of the application expert's view of the controlled system should not be used.

The requirements review process involves validating the relationship between changes in the real-world process and the specified changes and response in the control function model. Therefore, reviewability will be enhanced if the requirements specification explicitly shows this relationship. Moreover, when the description of the required controller behavior includes more than just its blackbox behavior (e.g., includes software design information and functional decomposition), then the semantic distance (d_4) between the process control behavior and the specified controller behavior increases, and the relationship between them becomes more difficult to validate. TCAS application experts who know very little or nothing about computers or software have been able to read our requirements model of TCAS and find errors in it.

In addition, a formal blackbox, behavioral model of the requirements makes possible (1) a mathematical verification of various desired properties such as consistency of the control model with the system goals and constraints, (2) the generation of standard system engineering and system safety analyses such as fault trees [Mel91] and (3) the application of formal correctness and robustness criteria to the specification model [JLHM91].

Although we believe that this type of blackbox specification is easier for application experts to review and easier to validate using formal analysis procedures, the semantic distance (d_5) between the requirements and a standard implementation based on functional decomposition is increased. To alleviate this problem, the specification step can be divided into separate requirements and design specifications or special software designs that result naturally from this type of blackbox specification may be used. If performance requirements can be satisfied, the specification can be implemented directly without an intervening design step.

Given the error-proneness of the requirements specification step and the few tools available to find these errors, the use of pure blackbox specifications (as advocated here and by Parnas et.al. [Hen80]) appears justified.

4 Specification Language

The first step in designing a specification language or modeling method is to determine goals and criteria for the language. This section describes general design criteria for such a requirements specification language and the language actually used to specify TCAS.

- Black-box
- Minimal
- Semantically simple
- Coherent, consistent, and concise
- Unambiguous underlying language with a formal foundation for analysis
- Readable, reviewable, and usable by application experts and developers
- Flexible notations (graphical, tabular, symbolic) tied to the best way to provide the particular type of information
- Readability given priority over writability
- User needs given priority over personal preferences
- Information exposure

Figure 3: Design criteria for the language

4.1 Design Criteria for the Specification Language

We identified several criteria that were important with respect to our goals and that we believe apply in general to this type of specification language (see Figure 3).

The first criterion, as described in the previous section, is that the language specify blackbox behavior of the software only and not include internal design information. Because of the safety and other types of formal analysis we planned to perform on the model, it also had to be based on a state machine as the underlying model; this is obviously not a requirement for all languages.

Two other criteria are minimality and simplicity. Minimality implies that the specification should contain only the information needed by the developers and analysts. Otherwise, time is wasted in specifying things that are not used. Many of the popular real-time requirements specification languages include facilities that are not strictly necessary. The problem with the "kitchen sink" approach is that the specification language becomes unnecessarily complex and the specification process becomes unnecessarily tedious and time-consuming. Also, for readability, information that is of limited help at a particular point in the specification should be omitted; the specification should help the reader focus on what is important.

To enhance simplicity, we tried to avoid specification language features that complicated the analysis and the specification. Language features that are semantically simple and straightforward to define are usually also easy to use and result in more readable and reviewable specifications.

Related to the minimality and simplicity criteria are coherency, consistency, and conciseness. Other specification languages for reactive systems, e.g., Statemate [HLN+90], Hatley/Pirbhai [HP87], and Ward/Mellor [WM85], include a variety of diverse models, some of which are not formally defined. Our goal was to specify all the required information using one formally-defined modeling language based on one underlying state-machine model. We also wanted our language to represent information as economically as possible while still maintaining readability.

Because of our goal to provide a safety analysis of the specification, the language must be unambiguous and the underlying model must have a mathematical foundation. At the same time, the requirements specification must be readable, reviewable, and usable. In some respects, these criteria may be conflicting but it is possible to satisfy both if there is a separation between the actual specification language and the underlying formal model. The specification must be unambiguous and translatable into mathematical notation, but it need not itself include arcane mathematical symbols that are unfamiliar to the application experts and software developers. We spent considerable time and energy developing a notation that was readable yet maintained the underlying formal state-machine model. This notation has graphical, symbolic, and tabular aspects depending on which was best for specifying a particular type of information [FG79]. Because readability and writability are often conflicting goals, we chose readability in cases where a conflict existed: The added investment in constructing the requirements specification pays off in terms of discovering more requirements-level errors.

The specification language was developed while specifying TCAS for the FAA, and we therefore received continual feedback by airframe manufacturers, component subcontractors, FAA certification experts, airline representatives, and pilot group representatives during development. This feedback provided invaluable information about the practicality, feasibility, and usability of the modeling language during its development. It helped us both with determining what did and did not need to be in the language and with satisfying our language design criteria.

One of the advantages of the feedback was to help us overcome our individual preferences. When devising the specification language, we usually had ourselves in mind as the user. However, our familiarity with certain notations, especially mathematical notations such as predicate calculus, hid their weaknesses. Our first attempts at devising our language, therefore, were failures: the notation was clear to us but not to others. The feedback from a diverse group of users helped us to evaluate the evolving specification language more objectively.

A final criterion for our specification became obvious only after trying to specify a complex system. We first used unrestricted hierarchical abstraction in our model, thinking this would aid in understanding the specification. We found that the use of what Harel [Har87] calls "clustering" (grouping states into superstates) indeed made the specification more readable. On the other hand, the use of what Harel calls "abstraction," a type of information hiding that allows showing only the superstate (as an empty state) and hid the component substates, often had an undesirable effect on readability. One of the purposes of such abstraction is that lower-level information, i.e., substates and transitions, can be hidden from the reader and in that way the system is presented in digestible chunks.

Our first modeling attempts maximized this type of hierarchical abstraction, thinking this would aid our goals of readability and understandability. Negative results were immediately apparent. Predicates (or guarding conditions) for transitions that "crossed" levels became very difficult to understand because they referred to nonvisible states. Context, which is vitally important to understandability, was lost. Thus, the information hiding concept that has contributed so much to the design, development, and maintenance of large, complex systems, proved detrimental to the understanding of such systems—a key element in requirements specification. For requirements specification, the reader (and specifier) needs as much context and specific detail as possible. We call this criterion "information exposure."

For the most part, our final TCAS specification has only two levels of abstraction—a top level to provide an overall global view and one lower level to model each major component in the controlled system. In a few places, a third level became necessary to aid understanding and ensure that each subcomponent model fit on one page. For TCAS II, this was all that was necessary, and we believe this to be true for most process control systems. The use of parallel state machines reduces the state explosion problem in state-machine models and each component of the process control loop usually has a limited number of relevant states and transitions.

4.2 Specification Language Description

Previously, we defined a formal state machine model called RSM (Requirements State Machine) for modeling the blackbox behavior of process-control systems along with formal criteria and heuristics to check the model for completeness, robustness, and safety [JLHM91]. RSM, while appropriate for formal analysis, has few of the desirable characteristics of a specification language. So we needed a usable specification language to put on top of the underlying RSM model.

Because our original goal was not to design a new specification language, we evaluated our criteria against existing languages, decided that Statecharts came the closest, and started specifying TCAS II using it. However, we soon realized that reviewers had difficulty understanding some aspects of pure Statecharts specifications and that some things we needed to specify were not easily described using it. Our specification language evolved as we got feedback on our drafts until it no longer is reasonable to refer to the language as Statecharts. We call our current formulation RSML (Requirements State Machine Language). This section describes the syntax and semantics of RSML and how it differs from Statecharts.

A basic state machine is composed of *states* connected by *transitions* (see Figure 4). *Default* or start states are signified by states whose connecting transition has no source. In the example, state A is the start state. Transitions define how to get from one state

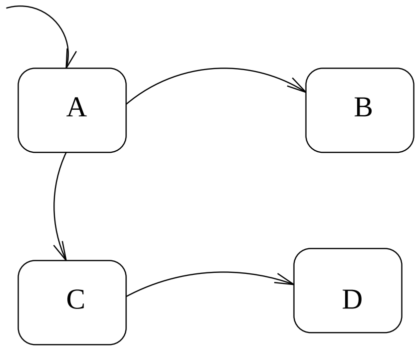


Figure 4: A Basic State Machine

to another. In the example, states B and C are directly reachable from A. State D is not directly reachable from A (no transitions connect the two states); however, state D is reachable from A via state C.

Statecharts are finite state machines augmented with hierarchy, parallelism, and modularity. An introduction to basic statechart notation can be found in [Har87]. RSML borrows the notions of superstates, AND decomposition, broadcast communication, statechart arrays, and conditional connectives from Statecharts. Other features of Statecharts, e.g., history and event selector connectives, were left out either because they were unnecessary or the semantics were too complicated to allow for formal analysis. We then added some features, such as interface descriptions and directed communication between state machines, and changed the syntactic notation to make it easier for our reviewers to read and review the specification. The syntactic extensions were found to be necessary to model a realistic problem rather than the small examples often found in research papers. We also changed somewhat the semantic definition of a "step," i.e., the semantics of state transitions. The rest of this section first describes the features in common with Statecharts and then our changes and extensions.

4.3 Features in common with Statecharts

Superstates. In Statecharts (and RSML), states may be grouped into *superstates* (see Figure 5). 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*). There are two ways to enter a superstate. First, the transition to the superstate may end at the superstate's border (transition A in Figure 5). 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 Figure 5). Note that the same superstate may have transitions ending at the border and at any number of the inner states. The superstate

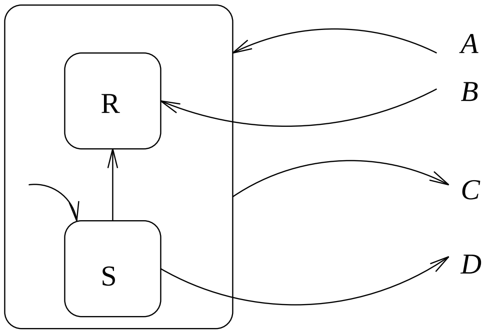


Figure 5: A superstate example.

may be exited in two ways (transitions C and D in Figure 5). 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.

AND Decomposition. One of the most important innovations in Statecharts is what Harel calls the *parallel state*[3] which contains two or more state machines, separated by dashed borders (Figure 6). 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 the TCAS system (i.e., the underlying RSM model) contains at least 10^{40} states whereas the graphical state diagram in our RSML specification of TCAS has approximately 100 states and fits on five pages. Although the syntax of parallel states is the same in both Statecharts and RSML, the semantic definition is different, as described in the Step Semantics section below.

Arrays. Both Statecharts and RSML allow the use of state-machine arrays (see Figure 7). State machine arrays are semantically equivalent to identical parallel state machines uniquely identified by an index. Each of the array elements is entered or exited when the array is entered or exited. Individual array elements are referenced by the array name and an index value. For example, Other-Aircraft[3] refers to the third array element in the example. We found that defining a special token 'THIS' that references the element value from within that element is useful for passing the identity of the element to a function, e.g., Traffic-Score(THIS).

Connectives. Conditional connectives are used when transitions out of a particular state into two or more different states are taken based on the same event but guarded by different

[3]Parallel states are also known as "orthogonal products", "product states", and "AND states".

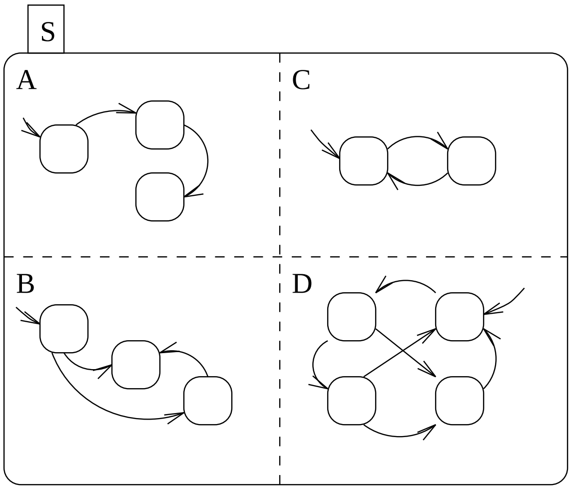


Figure 6: The parallel state

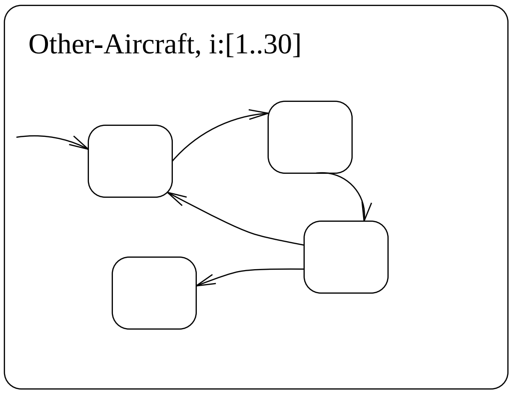
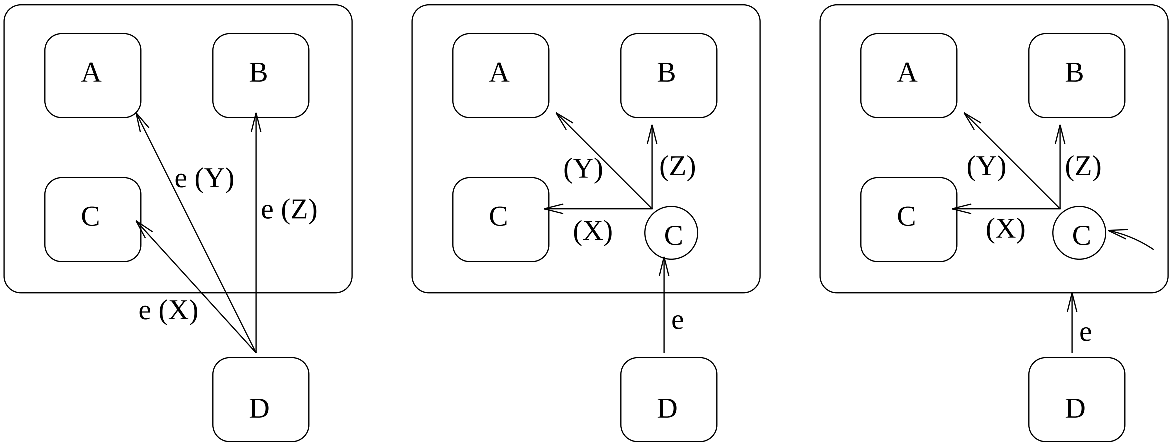


Figure 7: A State Machine Array



(a) State machine without a conditional connective

(b) Same state machine using a conditional connective

(c) Conditional connective used to select default state

Figure 8: In the diagrams, "e" is the triggering event and "X", "Y", and "Z" are the guarding conditions.

conditions (Figure 8 (a)). The transition from the source state to the connective is taken at the occurrence of the event. The appropriate destination state is determined based on guarding conditions that are defined on the transitions from the connective to the destination states (Figure 8 (b)). Some guarding conditions may be placed on the transition from source state to connective if all the destination states share those conditions. For a complete specification, the guarding conditions from the connective to the destination states must be mutually exclusive and must form a tautology [JLHM91]. Sometimes a state change is not desired. For these cases, a transition leads from the conditional connective back to the source state, thus explicitly specifying the circumstances for changing state *and* for remaining in a state.

A transition must begin and end in a state; therefore, the actual state transition is the transition from the source state to the connective combined with the transition from the connective to the destination state. Conditional connectives often appear as default "states" (Figure 8 (c)) in RSML, even though they are not states. The actual default state is chosen based on the conditions on the transitions out of the conditional connective.

4.4 Changes to Statecharts

Both syntactic and semantic changes and additions were made to these basic features of Statecharts.

Directed Communication. RSML includes the ability to model the behavior of all control loop components (not just the controller) and the communication between them. Physically distinct components are modeled as separate (communicating) state machines. Broadcast communication, as defined in Statecharts, is an inappropriate abstraction for communication between physically distinct components (e.g., two aircraft). Intercomponent communication in RSML is modeled as directed messages sent and received over unidirectional channels between component state machines. The limits of internal broadcast communication are denoted by thick borders around a component state machine (see Figure 9); internally broadcast events within a component state machine cannot cross thick borders.

Events. RSML includes two types of events—internal and external. Internal events are communicated within a single component state machine using the Statechart broadcast mechanism, i.e., they cannot cross the thick borders around the component state machines. Thus, TCAS does not necessarily know about any events in the altimeter or in other aircraft unless an external message has been sent between these two components. Internal events are used only for one very specific purpose: RSML specifications are pure blackbox specifications of the mathematical (input/output) function to be computed by the software; internal events are used to order the evaluation of that function. Basically they serve the same purpose as parentheses in algebraic equations.

External events, on the other hand, represent real communication (message passing) between TCAS and the other components (sensors, actuators, etc) of the system. They are required only because we include in our model the external interface to the system (in this case, TCAS) and the assumed behavior of the other components of the process control loop (the altimeters, other aircraft, pilots, etc.). The language does not prohibit the use of external events as triggering events on transitions; however, in the TCAS II specification, external event triggers are restricted to system component interface definitions.

Interface Definitions. The interface description is an important part of any requirements specification language. RSML includes an interface description for each separately modeled system component, which describes all external communication for that component. Our underlying model is communicating state machines: SEND events in one component trigger RECEIVE events in another component. Each communication specifies its source and destination. Unlike CSP [Hoa78] and some communicating state machine models, e.g., [Sha92], RSML does not require synchronous intercomponent communication.

The receipt of a message by a component state machine is signalled by the occurrence of an external RECEIVE event. These events may trigger state changes within the receiving component, i.e., values are assigned to input variables based on information communicated in the message. Because the state diagrams representing such state transitions are trivial and provide no useful information, only the transition descriptions are included in the RSML specification. The interface description includes the source and destination of the message, the triggering RECEIVE event and guarding condition, the mapping of message field names and values to variable names and values, and any internally generated events resulting from the receipt of the message. Note again that interface descriptions describe transitions within the receiving component state machine. Thus, guarding conditions will never block receipt of a message but may prevent the assignment of message field values to input variables.

Output variable value assignments and the sending of messages to other control-loop components are triggered by the occurrence of internal events. Each output interface description (representing a transition within the sending component state machine) contains the message source and destination, the internal triggering event and guarding condition, the mapping of output variable names and values to message field names and values, and the internally generated external SEND event.

Component State Machines. Each state machine is RSML may be divided into three parts separated by double solid lines (see Figure 9). The middle part contains the graphical state machine. The top and bottom parts contain input and output variables, respectively. All RSML inputs are blackbox inputs while outputs are calculated (derived) blackbox outputs.

Definitions must be provided for all input and output variables. Each definition contains:

- Location (the associated RSML state machine, e.g. Own-Aircraft).
- Source or Destination (external component, e.g. altimeter).
- Type (e.g., integer).
- Expected Range (e.g., -10,000 ... 10,000).
- Granularity (e.g., 10).
- Units (e.g., feet).
- Load (e.g., one per second).
- Exception Handling Information (e.g., out of range values are treated as zero).
- Traceability information (e.g., MOPS Reference).

In addition to the above items, output variable descriptions also contain triggering events and value assignments.

Transition Definitions. Transition definitions in RSML contain five parts: (1) the identification, (2) the location, (3) the triggering event, (4) the guarding condition, and (5) the output action. The identification, location, and triggering event are the only required parts. Figure 10 shows the form of a transition definition in RSML.

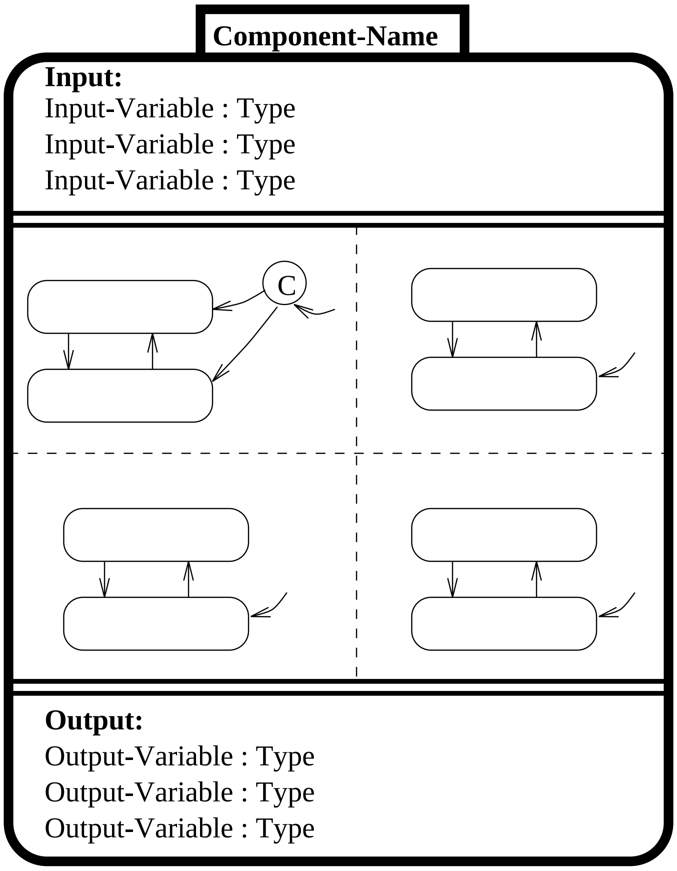


Figure 9: State Machine with Associated Variables

Each transition is identified by its source and destination states ($\boxed{\text{State-S}} \longrightarrow \boxed{\text{State-D}}$). Transitions split by a conditional connective are defined in two parts. The first part of the definition is identified by the connective destination, while the second part is identified by the connective source, ($\boxed{\text{State}} \longrightarrow$ © or © $\longrightarrow \boxed{\text{State}}$). If several transitions have the same definition (i.e., the same location, trigger, condition, and output action), then they may be defined together. Sometimes a single transition definition applies to all transitions into a particular state. The special symbol *ANY* may be used as a shorthand for all source states.

For example, the following transition identification

Transition(s): $\boxed{\text{North}} \longrightarrow \boxed{\text{East}}$
$\boxed{\text{South}} \longrightarrow \boxed{\text{East}}$
$\boxed{\text{West}} \longrightarrow \boxed{\text{East}}$

might be rewritten using:

Transition(s): $ANY \longrightarrow \boxed{\text{East}}$

The *location* field of the transition definition shows where in the state machine the transition may be found. The location is given as a hierarchical path, using the "$\triangleright$" symbol to separate the RSML state labels. For example, a transition at the location

Location: Other_Aircraft $\triangleright$ Tracked $\triangleright$ Intruder_Status$_{\text{s-52}}$

has its source and destination states in Intruder_Status (found on page 52) which is contained in the hierarchy formed by Other_Aircraft and Tracked. The location in the example can be traced in Figures 25, 28, and 29.

Transitions are taken upon the occurrence of the *trigger event*, provided that the guarding condition is true. Internally generated events may be either internal or external events, i.e. they are either broadcast within the component state machine or are explicitly sent to another component.

The *condition* defines what must be true before the transition can be taken and is specified using AND/OR tables, described below.

Output actions identify events that are generated when the transition is taken.

The rest of the transition definition is for explanation and documentation only. The *description* includes any English language description of the transition definition that may be appropriate to include and the *MOPS Ref.* is a reference to the pseudocode (design specification) that implements this transition. The latter provides traceability and was used in the independent verification performed on our TCAS specification. An optional *Comments* section can be used to provide extra explanatory information. For example, we sometimes used it to explain why a particular decision was made.

Transition(s): $\boxed{\text{(Source state)}} \longrightarrow \boxed{\text{(Destination State)}}$

Location: (path to the transition being considered.)

Trigger Event: (The event that causes this transition to be taken.)
Condition: (Optional guarding condition on the transition.)

Output Action: (Optional output action.)
Description: (Optional English description of the transition information)

MOPS Ref. (Used for tracing requirements to software design)

Comments: (Optional comments.)

Figure 10: Transition definition.

AND/OR Tables. Our first attempt to write the conditions for the state transitions used pure predicate calculus (Figure 11), as this was what we had seen in previous statechart examples [BGFG86, Har87] and it was natural to us. Our external reviewers, however, did not find it natural or reviewable and told to us to come up with something else. In fact, we found that we had difficulty in writing and reading complex predicate calculus expressions ourselves even though we were familiar and comfortable with the notation; while developing another notation, we found logical errors in our first attempt at specifying a part of TCAS that were not at all obvious in the original form.

Our second attempt replaced logical phrases with English phrases and a list of English-to-logic mappings. Although this is superficially more readable, we found that annotating the logic with English did not provide an appreciable advantage in terms of the underlying complexity of the logical expressions.

The notation we finally chose is a tabular representation of disjunctive normal form (DNF) that we call AND/OR tables.

AND		OR	
	Expression-1	T	T
	Expression-2	F	·
	Expression-3	·	T

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.

$$
\begin{array}{l}
\text{True-Tau-Capped}_{\text{f-362}} \geq \text{Time-To-CPA} \wedge \\
\quad (\ \text{Other-Capability}_{\text{v-212}} \neq \text{TCAS-TA/RA} \vee \\
\qquad (\ \text{Other-VRC}_{\text{v-209}} = \text{No-Intent} \wedge \text{Two-Of-Three}_{\text{m-327}})\ )\ \ \wedge \\
\quad (\ (\ \text{Down-Separation}_{\text{f-337}}(\text{low-firm}) \leq \text{Alt-Threshold} \wedge \\
\qquad\quad \text{Up-Separation}_{\text{f-362}}(\text{low-firm}) \leq \text{Alt-Threshold})\ \ \vee \\
\qquad (\ \text{Current-Vertical-Separation}_{\text{f-332}} > 150 \text{ ft} \wedge \\
\qquad\quad (\ (\ \text{Inhibit-Biased-Climb}_{\text{f-339}}(\text{low-firm}) > \text{Down-Separation}_{\text{f-337}}(\text{low-firm}) \wedge \\
\qquad\qquad\quad \text{Own-Tracked-Alt}_{\text{f-349}} < \text{Other-Tracked-Alt}_{\text{f-344}})\ \ \vee \\
\qquad\qquad (\ \text{Inhibit-Biased-Climb}_{\text{f-339}}(\text{low-firm}) \leq \text{Down-Separation}_{\text{f-337}}(\text{low-firm}) \wedge \\
\qquad\qquad\quad \text{Own-Tracked-Alt}_{\text{f-349}} > \text{Other-Tracked-Alt}_{\text{f-344}})\ )\ )\ )
\end{array}
$$

Figure 11: Transition Condition written in Predicate calculus

		OR					
	$\text{Other-Capability}_{\text{v-212}} = \text{TCAS-TA/RA}$	·	·	·	F	F	F
	$\text{Other-VRC}_{\text{v-209}} = \text{No-Intent}$	T	T	T	·	·	·
	$\text{Two-Of-Three}_{\text{m-327}}$	T	T	T	·	·	·
A	$\text{True-Tau-Capped}_{\text{f-362}} < \text{Time-To-CPA}$	F	F	F	F	F	F
N	$\text{Down-Separation}_{\text{f-337}}(\text{low-firm}) \leq \text{Alt-Threshold}$	T	·	·	T	·	·
D	$\text{Up-Separation}_{\text{f-362}}(\text{low-firm}) \leq \text{Alt-Threshold}$	T	·	·	T	·	·
	$\text{Inhibit-Biased-Climb}_{\text{f-339}}(\text{low-firm}) > \text{Down-Separation}_{\text{f-337}}(\text{low-firm})$	·	T	F	·	T	F
	$\text{Own-Tracked-Alt}_{\text{f-349}} < \text{Other-Tracked-Alt}_{\text{f-344}}$	·	T	·	·	T	·
	$\text{Own-Tracked-Alt}_{\text{f-349}} > \text{Other-Tracked-Alt}_{\text{f-344}}$	·	·	T	·	·	T
	$\text{Current-Vertical-Separation}_{\text{f-332}} > 150 \text{ ft}$	·	T	T	·	T	T

Figure 12: The AND/OR table

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. To make all these relationships clearer, we physically separated the columns, the far-left column a little more than the others. The AND/OR tables do not eliminate the need for existential and universal quantifiers; however, their scope is limited to a disjunct term or to the entire table, making it much easier to parse the expressions. We also discovered that omissions became apparent when application experts were forced to consider the explicit "don't cares" ($\cdot$) that appeared in the tables.

The above table is equivalent to

$$((\text{Expression-1} \;\wedge\; \neg\ \text{Expression-2}) \;\vee\; (\text{Expression-1} \;\wedge\; \text{Expression-3}))$$

The AND/OR table for the predicate calculus expression in Figure 11 is shown in Figure 12.

Some evidence of the readability and reviewability of the AND/OR tables is that errors we made in our first representation of the system were quickly discovered by the application experts after only a very minimal (ten minute) tutorial on our notation. Below the AND/OR tables, we later added an English language description of the guarding conditions on each transition.

Macros and Functions. As we wrote the TCAS requirements, we discovered that some of the AND/OR tables became very complicated. Also, some of the logic is repeated in several tables. We solved both problems by using macros, which are just labeled AND/OR tables. These macros, for the most part, correspond to typical abstractions used by the application experts in describing the TCAS requirements and therefore add to the understandability of the specification. We did, however, try to use them sparingly in order not to provide too many levels of indirection in the specification. To increase flexibility, macros may be parameterized. Also, rather than including complex mathematical functions directly in the transition tables, such functions are specified separately and referenced in the tables.

Transition Buses. One of the advantages of Statecharts over other state machine models is the ability to reduce a large number of states to a conceptually manageable number by using superstates and parallel states (AND-decomposition). We kept both of these features, but we found it helpful to introduce more constructs to reduce clutter. For example, many parts of the TCAS model are fully- or almost fully-interconnected, i.e., there is a transition from each state to nearly every other one. Showing each transition explicitly is confusing and can make the graphical diagram unreadable (see Figure 13A); the transition bus (Figure 13B) provides the same information. A transition must be defined for each source-state/destination-state pair on the transition bus, where a source state is a state with a transition to the bus and a destination state is a state with a transition from the bus.

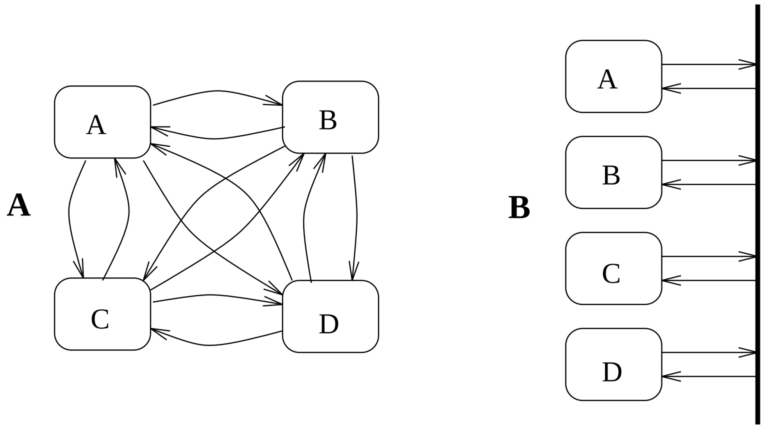


Figure 13:

Cross Referencing and Identifier Types. Another problem arose with writing transition information on the arrows between states. This is fine for relatively simple transitions and relatively simple statecharts. Even marking the arrows with a short tag that identifies the transition logic elsewhere was found to complicate the graphics and make it more difficult to comprehend when the statechart was complex. Such tags are symbolic "noise"; the information is not salient, even when using supposedly mnemonic tags, and the resulting clutter is more harmful than helpful. Unless the complete transition condition can be written on the arrow between states (not possible for anything but the trivial examples found in textbooks), such transition tags provide no useful information to the user except to match the arrow with a separate specification of the condition elsewhere in the document. The use of a special tag for this purpose merely increases the number of names and synonyms that the user must remember.

We opted instead, at first, to put page references on the arrow indicating where the transition logic could be found in the document (since this was really the information that the user needed); later we moved this information under each diagram in order to reduce clutter and make the diagrams more readable. This had no effect on the ease with which the page number information could be obtained. Paging through the document when reading transition definitions in order to view the corresponding statechart was minimized by including fold-out pages of the graphical part of the statecharts visible from anywhere in the document.

Cross referencing was used liberally elsewhere in the language as well. No matter how concise the notational style, requirements specifications for large systems span many pages (and sometimes volumes) and usually contain references to other parts of the specification. We wanted to reduce redundancy while still making easily accessible all information that is needed to understand or review each part of the document. Liberal use of page references as subscripts on names defined elsewhere was a practical compromise.

Another problem is how to identify the types of identifiers that are used in the specification. Solutions that have been used in the past include surrounding the name with special symbols as in the A-7 specification [Hen80] or using special fonts. Both of these solutions have drawbacks in terms of readability and learnability. The RSML approach is to use subscripts. Each identifier in the specification is subscripted with a single letter denoting its type—v for variable, s for state, m for macro, f for function, and e for event—and a page number where that element is defined. Page numbers are updated automatically when changes are made to the specification. For example, $altitude_{v-176}$ is a variable whose type definition can be found on page 176. An alphabetized index also is included that shows all the pages on which the name is used in the document and denotes the page on which the name is originally defined.

Identity Transitions. *Identity transitions* originate and terminate in the same state and generate output actions without causing a state change. The need for identity transitions arises when output actions are necessary for synchronization although no state change is required; the underlying Mealy machine model allows associating output actions only with transitions and not with states. All identity transitions are guarded by the negation of the disjunction of all the conditions guarding transitions that are triggered by the same trigger event and that originate in the same state. Identity transitions are not included in the state-machine graphical diagram in order to reduce clutter since they are not needed for reviewing the specification but for analysis and completeness reasons. Instead, they are grouped in tables.

Timing. During the specification of TCAS, we needed only three temporal functions: the value of a variable at some previous point in time, the truth value of a condition at some point in the past, and an implicitly generated event based on time (i.e., a timeout relative to state entry). Rather than treating timeouts specially and defining triggering events for them, all timing functions are written as expressions in guarding conditions. An example of such an expression is $\boxed{t \geq (t(\text{entered(Threat)}) + 5.0 \text{ secs})}$. This expression states that the current time ('t' without an argument) is greater than or equal to the time that state Threat was entered plus 5.0 seconds. Such expressions evaluate to true or false and generally appear as logical phrases in AND/OR tables.

TCAS is required to operate based on a cycle, called a surveillance cycle, started by an event (Surv-Comp-Event) and all temporal requirements are based on this cycle. In the TCAS requirements document the function $\textsc{Prev}_j(\text{x})$ has been overloaded to apply to both variable values, functions, and predicates:

- $\textsc{Prev}_j(v)$ refers to the value of variable (or function) v at j surveillance cycles back in time.

- $\text{PREV}_j(p)$ refers to the truth value of p at j surveillance cycles back in time.

Step Semantics. The semantics of Statecharts have been described in detail in several papers [HP85, Har87]. Unfortunately the descriptions are not consistent with each other; small (but significant) differences exist. The following comparison between Statecharts and RSML semantics is based on the formal description by Pnueli and Shalev [PS89].

The semantic description of Statecharts in [PS89] is based on the notion of steps. A step is initiated when an external event arrives at the model boundary, causing a cascade of subsequent internal events. A step is completed when no more internal events are generated or there are no more transitions triggered by the events that were generated, i.e., the model has stabilized in a state. It is assumed that a step is completed before another external event arrives, i.e., there is no delay in the response to an external stimulus (this assumption is called the *synchrony hypothesis*). The main difference between Statecharts and RSML is in the way a step is constructed.

In both Statecharts and RSML, a step starts when a set of external events (I) arrives at the model boundary. The set of transitions that are triggered by the events in I are denoted by $triggered(I)$. If a transition is taken, a new event may be generated. The set of events generated when the transitions in a set of transitions T are taken is defined by $generated(T)$. The state (or configuration) of the model is denoted by C, e.g., the initial configuration in Figure 16 is the set $\{A, C\}$. Transitions whose source states are members of a particular configuration C are said to be *relevant* to C.

In the set of transitions denoted by $relevant(C)$, a (possibly empty) subset are triggered by the events in I, i.e., the transitions that could possibly be taken given a configuration C and a set of events I are defined by

$$relevant(C) \cap triggered(I).$$

In this set, only a few transitions are compatible, i.e., can be taken together. For example, the transition labeled a in Figure 14 is not consistent with the transition labeled b since only one of them can be taken. Let the set $consistent(T)$ contain all transitions that are compatible with the transitions in the set T.

The construction of a step in Statecharts is based on an enabling function En that determines which transitions can be taken given a set of external events (I) and a configuration C. In the Statecharts step creation, a transition relevant to C and triggered by an event in I is initially picked and added to the set T. T is then expanded by adding transitions that are relevant to C, consistent with the transitions already in T, and triggered by either an event in I or an event generated by a transition in T. For example, consider the model in Figure 16. When the external event x arrives and the model is in the initial configuration $\{A, C\}$, t_1 and t_4 are the only two transitions that can be taken. Assume t_1 is picked and added to T. Since t_1 generates the event y, the new set of possible transitions

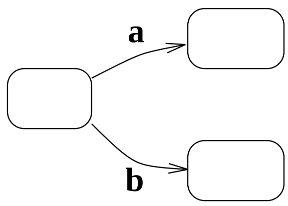


Figure 14: Transitions a and b are inconsistent

Procedure Step-Construction(**Var** C : configuration, I : setofevents) ;
begin
  T := $\emptyset$;
  while T $\subset$ En(T, C, I) **do**
    nondeterministically pick a transition t $\in$ (En(T, C, I) $-$ T) and add it to T ;
  C := NextConfig(C,T) ; { Calculate the new configuration }
end

Figure 15: Step construction in Statecharts

is expanded to include t_3. Both t_4 and t_3 are consistent with t_1, so either one can be picked and added to T.

In general, the set of transitions that can be added to T can be defined with an enabling function:

$$\begin{aligned} En(T,C,I) &= relevant(C) \cap \\ &\quad consistent(T) \cap \\ &\quad triggered(I \cup generated(T)). \end{aligned}$$

The construction of a Statechart step is defined by the operational definition in Figure 15.

The function $NextConfig$ calculates the new state configuration given the old state configuration C and the set of transitions T. The possible constructions of a step in Figure 16 are summarized in Table 1. The configuration at the beginning of the step is defined by the set $\{A, C\}$, assuming that $I = \{x\}$. Note here that due to the nondeterministic nature of the step construction (i.e., the selection of the transition to put in T is made nondeterministically), there are (in this case) three different ways of constructing a step; two constructions yielding different results are illustrated in the table. The behavior defined in construction 1 in Table 1 is counterintuitive since transition t_4, which should "obviously" be triggered by the input event x, is not taken.

The semantics of RSML is slightly different and enforces a more rigorous causal ordering of the transitions taken within a step. The enabling function in the RSML step construction

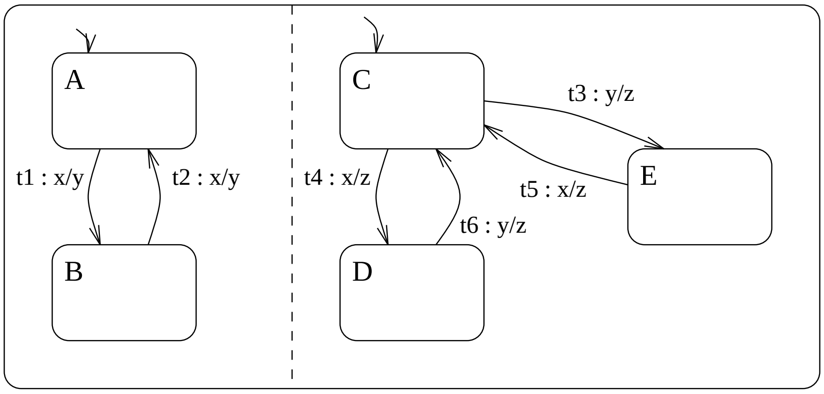


Figure 16: A Statechart and RSML example

Construction 1				
loop #	**T**	**En(T)**	**generated(T)**	**C**
0	$\emptyset$	$\{t_1, t_4\}$	$\emptyset$	$\{A, C\}$
1	$\{t_1\}$	$\{t_1, t_3, t_4\}$	$\{y\}$	$\{A, C\}$
2	$\{t_1, t_3\}$	$\{t_1, t_3\}$	$\{y, z\}$	$\{A, C\}$
Completion of Step				$\{B, E\}$

Construction 2				
loop #	**T**	**En(T)**	**generated(T)**	**C**
0	$\emptyset$	$\{t_1, t_4\}$	$\emptyset$	$\{A, C\}$
1	$\{t_4\}$	$\{t_1, t_4\}$	$\{z\}$	$\{A, C\}$
2	$\{t_1, t_4\}$	$\{t_1, t_4\}$	$\{y, z\}$	$\{A, C\}$
Completion of Step				$\{B, D\}$

Table 1: Two possible step constructions in Statecharts

```
Procedure Step-Construction-RSML(Var C : configuration, Var I : setofevents) ;
begin
    repeat
        T := ∅ ;
        while T ⊂ En(T, C, I) do
            nondeterministically pick a transition t ∈ (En(T, C, I) − T) and add it to T ;
        C := NextConfig(C,T) ; { Calculate the new configuration }
        I := generated(T) ; { Calculate the internal events generated by the transitions
                              in T and use them to continue the construction of the step }
    until T = ∅ ;
end
```

Figure 17: Step construction in RSML

does not consider the transitions triggered by the output events of the transitions in T to be enabled, i.e.,

$$\begin{aligned} En(T,C,I) &= relevant(C) \cap \\ &\quad consistent(T) \cap \\ &\quad triggered(I). \end{aligned}$$

The step construction in RSML can now be described by the algorithm in Figure 17.

This definition forces an RSML state machine to take first all transitions triggered by the external event starting the step and then the transitions triggered by the events generated as a result of that first *micro-step.* The process is repeated until there are no more transitions triggered by the events generated by the preceding micro-step. Table 2 shows the construction of a step according to the semantics of RSML.

This approach has one disadvantage compared to the Statecharts step construction. It can easily be seen that the Statecharts step construction will always terminate since $En(T)$ is a finite set. The step construction in RSML state machines can potentially be infinite as is shown in Figure 18 and Table 3. The events in Figure 18 will get alternately generated forever.

The advantage of the RSML approach is that it is more consistent with our intuitive notion of a step. In the example shown in Figure 16, the Statecharts step semantics allows transition t_3 to be taken, even though event y is generated after x. A reviewer could be misled by such a specification, not realizing that the specification is inconsistent with what is intended. We felt that reviewability, in this case, was more important than the ability to force termination.

outer loop #	**inner loop #**	**T**	**En(T)**	**I**	**C**
1	0	$\emptyset$	$\{t_1, t_4\}$	$\{x\}$	$\{A, C\}$
1	1	$\{t_1\}$	$\{t_1, t_4\}$	$\{x\}$	$\{A, C\}$
1	2	$\{t_1, t_4\}$	$\{t_1, t_4\}$	$\{x\}$	$\{A, C\}$
1	Exit	$\{t_1, t_4\}$	$\{t_6\}$	$\{y, z\}$	$\{B, D\}$
2	0	$\emptyset$	$\{t_6\}$	$\{y, z\}$	$\{B, D\}$
2	1	$\{t_6\}$	$\{t_6\}$	$\{y, z\}$	$\{B, D\}$
2	Exit	$\{t_6\}$	$\emptyset$	$\{z\}$	$\{B, C\}$
3	0	$\emptyset$	$\emptyset$	$\{z\}$	$\{B, C\}$
Exit	—	$\emptyset$	$\emptyset$	$\{z\}$	$\{B, C\}$

Table 2: The step construction in RSML

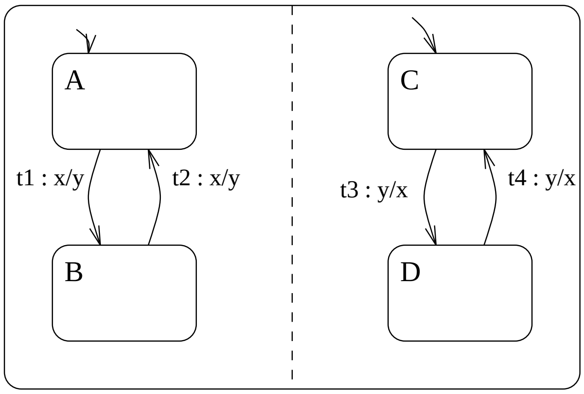


Figure 18: An RSML state machine that will not terminate

outer loop #	**inner loop #**	**T**	**T'**	**En(T)**	**I**	**C**
1	0	$\emptyset$	$\emptyset$	$\{t_1\}$	$\{x\}$	$\{A, C\}$
1	1	$\{t_1\}$	$\emptyset$	$\{t_1\}$	$\{x\}$	$\{A, C\}$
1	Exit	$\{t_1\}$	$\{t_1\}$	$\{t_1\}$	$\{y\}$	$\{B, C\}$
2	0	$\emptyset$	$\{t_1\}$	$\{t_3\}$	$\{y\}$	$\{B, C\}$
2	1	$\{t_3\}$	$\{t_1\}$	$\{t_3\}$	$\{y\}$	$\{B, C\}$
2	Exit	$\{t_3\}$	$\{t_3\}$	$\{t_3\}$	$\{x\}$	$\{B, D\}$
3	0	$\emptyset$	$\{t_3\}$	$\{t_2\}$	$\{x\}$	$\{B, D\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3: An example of a RSML state machine leading to an infinite step construction.

5 The System Requirements Specification

The RSML language was developed using TCAS II as a testbed. The resulting specification not only acts as an example of a blackbox process-control system requirements specification, but also as an example of a real-life, successful application of formal methods to a complex system. We caution the reader, however, that compromises in the specific model created for TCAS II were required due to real-life constraints explained further in Section 6 (Evaluation and Future Goals).

Figure 19 shows the contents of the TCAS specification[4] (all figures in this section are located at the end of the paper). There are similarities in content with the A-7 requirements document [Hen80], but we include behavioral descriptions of the other components in the process control loop as well as system goals and constraints. Physical requirements for the TCAS box (e.g., size, weight and materials) and some I/O devices (e.g., TCAS antennas) are contained in the MOPS and should be in our document, but currently are not (simply due to a lack of resources to retype them). The A-7 specification includes sections on possible subsets of the program and the characteristics of the computer, which we do not include. Both specifications include requirements for timing, accuracy, and response to undesired events, but we do not separate them from the functional behavior; instead they are included where the functional behavior is specified. Jaffe [Jaf88] has argued that functional and timing information is too inextricably connected to be usefully separated.

The goals and constraints in the first section of the document are written in English. In general, they would be the first thing specified when developing the system, although they may be modified as the system engineering process progresses.

Normally, the next step in system engineering involves identifying and designing (if necessary) the components of the control loop. In the case of TCAS, most of the components already exist. Early in the system design process, a detailed description of the allocation of high-level functional requirements to each physical component along with the interfaces between them is generated.

The environment section of the specification includes a high-level system component and communication diagram (see Figure 20). Note that this diagram is a directed graph and not a state machine diagram. The TCAS system consists of various sensors (e.g., radio altimeter and Mode-S transponder) and actuators (e.g., a pilot display and transmitters) aboard the aircraft. Some of these communicate with other aircraft (which have varying collision avoidance capabilities) and ground radar stations.

The black-box behavior of each control loop component (except TCAS) and the relevant behavior of each process component is modeled in the environment section. The RSML specifications of the physical components other than TCAS itself reflect the assumptions

[4] The entire TCAS specification is complete at this time except for Section 3.1. Only a few of the environment components have been specified so far due to FAA pressures to deliver the TCAS specification first. Section 3.1 is primarily required only for the safety analysis.

that the designers of TCAS can make about the components' behavior including their failure behavior. Including these assumptions in the specification is useful in designing the controller software to be robust against the effects of design changes to the other components and against failures in the environment.

As an example, Figure 21 shows the RSML state machine description of the radio altimeter. This device provides CAS with own (the host) aircraft's altitude above the ground. An accompanying status message indicates, to some extent, the reliability of the altitude data based on an altimeter self-test mechanism. Under normal operating conditions, the radio altitude is correct (within a certain tolerance) and the status indicates "okay." In one possible failure mode, the radio altitude is correct, but the status indicates a failure. In a second failure mode, the radio altimeter produces no output, neither altitude data nor status information. Finally, in the third failure mode, the radio altitude sent is incorrect, but the status indicates "okay." In this mode, the altimeter may send all zeroes, repeatedly send the maximum, repeatedly send the same value, or send random values.

The environment section also includes the description of the communication interface between the components of the control loop. This includes CAS inputs and outputs although other communication (such as between other aircraft) would be included if it were relevant to the operation of TCAS. Figures 22 and 23 show examples of an input interface and output interface specification, respectively. Figure 24 shows an example of a message formal specification.

TCAS has three logical subcomponents (see Figure 20): the collision avoidance subsystem (CAS) that contains the actual collision-avoidance logic, a surveillance subsystem that handles communication with other aircraft and the ground radar stations, and a performance monitor. All three could have been specified together as one logical component, but for historical and political reasons we were required to specify the behavior of each separately. The three models are therefore specified as separate components with defined external interfaces. CAS is by far the most complex of the three and is used here as the example. The following overview of the CAS specification guides the reader through examples of the various parts of the specification although the description is greatly simplified in order to make it understandable to those unfamiliar with collision avoidance systems.

The highest level CAS state machine is shown in Figure 25. At this level, CAS is either on or off; if it is on, it may be either fully operational or in standby mode. As explained previously, the control function is specified only in terms of the state of the controlled process and the states of relevant control loop components. In the case of the CAS logic, the states of three types of process components are modeled: our own aircraft, other aircraft, and mode-S ground radar stations. Each of the three subcomponents of CAS is elaborated in more detailed RSML models.

Figure 26 shows the expanded Own-Aircraft portion of the CAS model. The top portion of the diagram lists variables that represent inputs to CAS from the TCAS sensors that are associated with the state of Own-Aircraft. The bottom portion of the diagram lists variables that represent outputs from CAS to TCAS actuators. The middle portion of the diagram represents the parts of the derived Own-Aircraft state necessary for the evaluation of the CAS control function.

Effective-SL (sensitivity level) controls the dimensions of the protected airspace around own aircraft. It is used to control the trade-off between necessary protection and unnecessary pilot advisories. Higher sensitivity levels increase protection, but also increase the incidence of unnecessary alerts.

There are two primary means that CAS uses to determine Effective-SL: ground-based selection and pilot selection. Ground-based selection of sensitivity level is not envisioned for use in the U.S. airspace at this time; however, the capability for such selection has been included in the CAS logic. The pilot, on the other hand, can select three modes of operation (STANDBY, TA-ONLY, and TA/RA) which are converted to sensitivity level by the logic. In STANDBY mode, neither traffic advisories (TA's) nor resolution advisories (RA's) are output by CAS. The pilot normally selects STANDBY when on the ground. In TA-ONLY mode, only traffic advisories are output by CAS. This mode is often selected by the pilot to avoid unnecessary distractions while at low altitudes on final approach to an airport. When the pilot selects TA/RA mode (also called AUTOMATIC), CAS selects sensitivity level based on the current altitude of own aircraft (Auto-SL state).

Alt-Layer effectively divides vertical airspace into layers (e.g., Layer-3 is approximately equal to the range 20,000 feet to 30,000 feet). State changes are made using a hysteresis: the criteria for transitioning into the Layer-2 state is different depending on whether the current state is Layer-1 (own aircraft is climbing) or Layer-3 (own aircraft is descending). Alt-Layer and Effective-SL are used in the determination of individual other aircraft threat classification (see Figure 29).

Due to aircraft climb performance limitations at high altitude or in the landing configuration, the CAS logic may inhibit a climb maneuver. Descend maneuvers are inhibited if own aircraft is too close to the ground to safely command the pilot to descend. The increase inhibits (Increase-Climb-Inhibit and Increase-Descend-Inhibit) prohibit the command of higher rate maneuvers (e.g. 2500 fpm vs. 1500 fpm), and therefore use more stringent altitude thresholds. The Advisory-Status part of the Own-Aircraft model (Figure 27) shows the CAS resolution advisory (RA), if there is one, that is currently displayed to the pilot.

Figure 28 shows the expanded Other-Aircraft portion of the CAS model. Again, the top portion of the diagram lists variables that represent inputs to CAS from the TCAS sensors, and the bottom portion of the diagram lists variables that represent outputs from CAS to TCAS actuators. The middle portion of the diagram contains parallel state machines representing the derived Other-Aircraft state necessary for the evaluation of the CAS control function.

RM-Send-Status synchronizes coordination interrogations with other TCAS-equipped aircraft, where "RM" stands for Resolution Message. Coordination interrogations contain information about an aircraft's intended vertical maneuver or "intent" with respect to a threat. This information is expressed in the form of a complement; e.g., if own aircraft has selected a climb maneuver against the threat (see Figure 29), it will transmit a message in its coordination interrogation restricting the threat aircraft to descend maneuvers against own aircraft.

CAS can track up to 30 aircraft simultaneously (it can track more but is limited by the number of conflicting flight scenarios it can resolve simultaneously). The Track-Status state reflects whether a particular Other-Aircraft is currently being tracked or not. Figure 29 shows the expanded Tracked portion of the Other-Aircraft RSML model.

The Intruder-Status state within Tracked reflects the current classification of Other-Aircraft (Other-Traffic, Proximate-Traffic, Potential-Threat, and Threat). Intruder-Status is determined using (among other criteria) Own-Aircraft Effective-SL and Alt-Layer. 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.

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. Possible advisory strengths are Nominal-1500fpm (1500 feet per minute), VSL-2000, 1000, 500, and 0-fpm (vertical speed limits of 2000, 1000, 500 and 0 feet per minute; 0-fpm means level flight). Advisory strength is continuously evaluated and modified, if necessary, during the course of the encounter. After CAS has chosen an RA, occasionally the threat aircraft maneuvers vertically in a manner that thwarts the RA. In this case, CAS may increase the strength of the advisory from 1,500 feet per minute to 2,500 feet per minute (Increase-2500fpm) or it may reverse sense (from Climb to Descend or vice versa).

The Mode-S-Ground Station model is quite simple and, therefore, is not shown here. Although theoretically the CAS logic uses input from the ground stations, these are not operational at this time.

The specification must include a description of each input and output variable. Examples are shown in Figures 30 and 31.

As an example of a transition definition, Figure 32 contains the definition of the transition from the state Threat to the state Other-Traffic, substates of Intruder-Status. In order for an intruder to be classified as a Threat, it must be reporting its altitude, it must be airborne, and it must satisfy the threat altitude and threat range tests. Once classified as a Threat, it may not be downgraded (to Potential-Threat, Proximate-Traffic, or Other-Traffic) based on the threat altitude test. It may be downgraded based on the threat range test, but only if it fails on two consecutive attempts—represented by a separate transition. However, if the intruder stops reporting altitude, or if it reports that it is on the ground, it can no longer be classified as a threat. The first and last rows of the AND/OR table represent this criteria. Note that this transition represents a downgrade directly to Other-Traffic, bypassing the intermediate classifications of Potential-Threat and Proximate-Traffic. This happens when the intruder is no longer airborne (column 4) or when altitude reporting is lost and either the bearing or range inputs are invalid (columns 1 and 2). Column 3 represents a situation in which a partial downgrade to Potential-Threat or Proximate-Traffic might have been possible, i.e., altitude reporting is lost, but both the range and bearing inputs are valid. If either the Potential-Threat-Condition or the Proximate-Traffic-Condition were satisfied, the intruder classification would have been downgraded to Potential-Threat or Proximate-Traffic, respectively. However, in this transition, neither criteria are satisfied, so the classification is downgraded all the way to Other-Traffic.

As an example of a macro, Figure 33 contains the Potential-Threat-Condition macro referenced in the above transition. In order for an intruder to be classified as a Potential-Threat, it must satisfy the Potential-Threat-Range-Test. In addition, if it is reporting altitude and is airborne, it must satisfy the Potential-Threat-Altitude-Test. If it is not reporting altitude, own aircraft must be below 15,500 feet.

Functions and macros are used in a similar way, but functions return values. Figure 34 contains an example of an RSML function, Vertical-Resolution-Complement. This function is related to the coordination interrogations described earlier (Other-Aircraft ▹ RM-Send-Status). If CAS has selected a climb maneuver against this particular intruder (Other-Aircraft in state Climb), the Vertical Resolution Complement (VRC) is Don't Climb. A value of 2 will be assigned to the VRC field of the Mode S message.

The appendices to the document contain additional information to make the specification more readable or changeable. The first appendix defines constants. Everywhere a constant is used in the document, a label is attached as a subscript, e.g., $300ft._{(MINSEP)}$ associates the constant 300 feet with a label that designates this is the minimum vertical separation allowed between aircraft. A change of the value of this constant (e.g., the FAA decides in the future that minimum vertical separation should be 350 feet) can be automatically and easily made throughout the document. An alternative (and more common) solution to the maintenance and change problem for constants would be to use the label alone throughout the document and put the values associated with the labels into a table. However, the latter solution makes the document much less readable and requires constant flipping to the constant definition section to determine the actual numbers associated with the labels.

The second appendix, Table Definitions, is used for constants that are more naturally stored in a tabular form, e.g., potential-threat minimum-range threshold indexed by our own aircraft sensitivity level.

We found that a list of events associated with the state transition that generates them and the state transitions that are triggered by them was helpful in producing the document and included this list in a third appendix.

The Glossary contains definitions of technical terms and abbreviations used throughout the document, and the Notation Appendix provides a tutorial on the RSML language.

The Reference Algorithms appendix contains tracking and other algorithms that are not required but are used to define criteria for accuracy of the actual algorithms selected. For example, a tracker chosen by the designer might be required to have at least the accuracy of the alpha-beta tracker specified in the appendix.

Finally, an index to the document is provided that includes an entry for every name used in the document giving the pages on which it is used and the page where it is defined. Currently, there are over 500 entries in the index.

6 Evaluation and Future Goals

This paper has defined (1) an approach to specifying system requirements for real-time, reactive systems, (2) the criteria that should be used in designing a language for such requirements, (3) a language demonstrating the approach and criteria, and (4) the necessary and desirable contents and organization of a system requirements specification using this approach. These were developed while writing a system requirements specification for an aircraft collision avoidance system, which provided continual evaluation and feedback during development and demonstrated the practicality of writing a formal requirements specification for a complex process-control system.

Because the specification (which was originally intended merely to be experimental) was adopted by the FAA during the development of RSML, deadlines required us to deliver parts of the specification while the notation was still evolving. There are some aspects of this type of a specification that still cause difficulties in understanding such as the overall event sequencing and synchronization. During the independent verification and validation of our TCAS specification, we needed to derive addition diagrams and tables so that the reviewers could easily check the consistency of our specification with the previous, pseudocode version. Although parallel state machines and other features of the language did reduce the specification of states enough to make such state-machine specifications practical, the proliferation of events causes problems that need to be handled.

Reviews of our document for correctness by users during development made clear that specifications should include graphical, symbolic, tabular and textual notation, depending on the type of information being conveyed. For example, the graphical state machines were a great help during reviews for finding certain types of errors as were the tables for finding other errors. Even though the state transition information had to be removed from the graphical state diagrams and put into tables, the graphical representation provided important information to reviewers of the document that would have been very difficult or impossible to derive from the transition tables alone. A language that contains only graphics or only tables or only symbolic strings is probably less useful than one in which different notational techniques are used to communicate different types of information. More research is needed to determine the most appropriate notations for each type of information that needs to be conveyed.

One result of this effort was a demonstration that formal specifications can be applied to complex, reactive systems and that such specifications can be readable and reviewable by application experts with a minimal knowledge of mathematics and computer science. A lesson to be learned from the experience is that formal specifications can be usable if their design takes into consideration the training and backgrounds of those who are to read and review the specification. Some engineers working with us on the TCAS specification reported that they liked the AND/OR table description of the transition conditions because it resembles the logic tables that they are used to using and that the state machines and logic tables fit the way they think about systems.

Although formal specification languages obviously have to be defined in an unambiguous and mathematical way, the syntax itself does not have to contain obscure mathematical symbols that are familiar and comfortable to neither the application expert nor the implementor of the system. There must simply be an unambiguous translation from the specification language (in our case RSML) to the formal model (RSM for our language) underlying it. Currently, formal specification languages are designed primarily by mathematicians who use a notation with which they are comfortable, but which is foreign to those who must use the language. One solution is to train hardware and software engineers to think like mathematicians while our alternative solution is to provide languages that allow the user to think about the system in the way that they have been trained in their discipline. We hypothesize that providing a model of a system that is closer to the mental model that the reviewer and implementor have of the system and closer to the way they have thought about such systems in the past will aid in finding errors in the specification itself and reduce the numbers of errors that are introduced in implementing the specification. This hypothesis, of course, still needs to be experimentally validated, although our experience provides some anecdotal support.

Because the specification of the CAS logic, from which we built the CAS part of our TCAS model, was low-level pseudocode, the exercise had many features of reverse engineering. The pseudocode used is a low-level language containing only:

- simple data types (bits, bit strings, character strings, integers, pointers, and floating point variables),
- arithmetic expressions,
- the structured control statements IF-THEN-ELSE, IF-ELSEIF-OTHERWISE, REPEAT-WHILE, REPEAT-UNTIL, and LOOP-EXITIF-LOOP, and

- subroutines (without local variables).

All variables are global: There are no local variables but there is provision for passing parameter names to subroutines to show which variables are used by the subroutine (few subroutines actually use this feature in the TCAS specification). The only complex data structure allowed is a "group" that provides for grouping related variables into a "data structure," i.e., giving them a group name.

In many ways, the TCAS reverse engineering was even more difficult than the usual reverse engineering exercise since the language was so low-level and difficult to read. This specification has acted as the requirements specification for TCAS from 1983 to 1992 and was continually changed as errors were found and changes made to the requirements. Several lessons can be learned from our experience that are applicable to both forward and reverse engineering efforts in general.

First, we had difficulty abstracting away from the design. Even when we did not look at the pseudocode, we found it difficult in the beginning to eliminate functional decomposition and flowchart-like logic, i.e., to specify the problem without trying to solve it. With practice we became better at omitting design information, but the struggle never entirely abated. The very low level of the pseudocode also made the process of abstraction more difficult as many purely implementation features, such as flags, had to be used extensively in the pseudocode. After the specification of the CAS logic was completed, an independent verification and validation was performed to compare the pseudocode specification and the RSML specification. The verifiers experienced the same problems that we did, and a large number of identified discrepancies resulted in no change to the RSML specification because they merely represented design peculiarities of the pseudocode and not requirements.

Second, although it may be a function of the particular system we were reverse engineering, we found it impossible to derive the requirements specification strictly from the pseudocode and an accompanying English language description. Although the basic information was all there, the intent was missing. Therefore, distinguishing between requirements and artifacts of the implementation was not possible in all cases. As has been discovered by most people attempting to maintain such systems, an audit trail of decisions and the reasons why decisions were made is absolutely essential. This was not done for TCAS over the 15 years of its development and those responsible for the system today are currently attempting to reconstruct decision-making information from old memos and corporate memory.

Third, the final requirements specification model would have been different and much simpler if we had been starting from scratch. Because the TCAS pseudocode specification had evolved over a period of more than 15 years, the current version contains more complexity than is necessary. What was originally a simple conceptual model degraded as changes were made to the pseudocode that simplified the process of making the change or minimized the amount of code that needed to be changed, but complicated or degraded the original conceptual model. As Parnas said in [Par79]: "The problem is that the subsets and extensions are not the programs that we would have designed if we had set out to design just that product." This is a common maintenance dilemma, and TCAS was no exception. When changes are made to design or code without backing up all the way to requirements, such problems arise and increase as time passes. For TCAS, the highest-level specification *was* the pseudocode.

The problem of increasing complexity and lack of conceptual coherency in the underlying model were exacerbated as more and more changes were made over the years and more errors introduced due to the increasing difficulty in determining the consequences of the changes. What we did for the TCAS system was to make the current underlying conceptual model explicit. Because of the necessity of building a requirements specification that matches the TCAS systems actually in use (which were certified against the pseudocode specification), our resulting model is more complicated than necessary, includes more than the minimum required behavior, and is harder to understand than is strictly necessary. This was frustrating as we first built a nice, simple model and found that we had to complicate it for no better reason than that it had to match some errors or poor decisions in the pseudocode. Once our specification is complete, future versions of the system will hopefully return to a simpler model. We believe that if a blackbox behavioral model of our type had been built originally, not only would the final specification be simpler and more understandable, but making changes without introducing errors or unnecessarily complicating the resulting requirements also would have been simplified.

Now that the specification is complete, our work on validating the feasibility and practicality of formal analysis procedures on such specifications has begun. Heimdahl [Hei94] has (1) implemented a simulator for RSML so specifications can be executed, (2) formally defined the semantics of RSML using composable functions, (3) devised algorithms to perform semantic analysis on the underlying RSM formal model to ensure completeness and consistency in requirements [JLHM91], and (4) experimentally validated the analysis algorithms on the TCAS II specification.

We are currently working on analysis procedures (1) to analyze the entire system model for safety [LS87], and (2) to perform standard system engineering risk analyses such as fault tree analysis [Mel91] directly from the system requirements specification. Attempts have also begun to derive test data satisfying various coverage criteria automatically from the specification [WGS94].

Acknowledgments

Important contributions to this effort were made by Ruben Ortega and Rueven Greenberg, both of whom were graduate students at UCI. We would also like to acknowledge the help of Mike DeWalt, Jim Treacy, Larry Nivert, and Tom Choyce of the FAA and the members of the RTCA Working Group on TCAS Requirements, especially Kathryn Ybarra of Honeywell, David Lubkowski and Uma Satyen of MITRE, Gus Kyriakos of Bendix, Amy Johnson and Jose Perez of Rockwell Collins, Ann Drumm of Lincoln Labs, and Captain Ross Beins (United Airlines) of the TCAS Pilots Working Group.

Figure 19: The table of contents

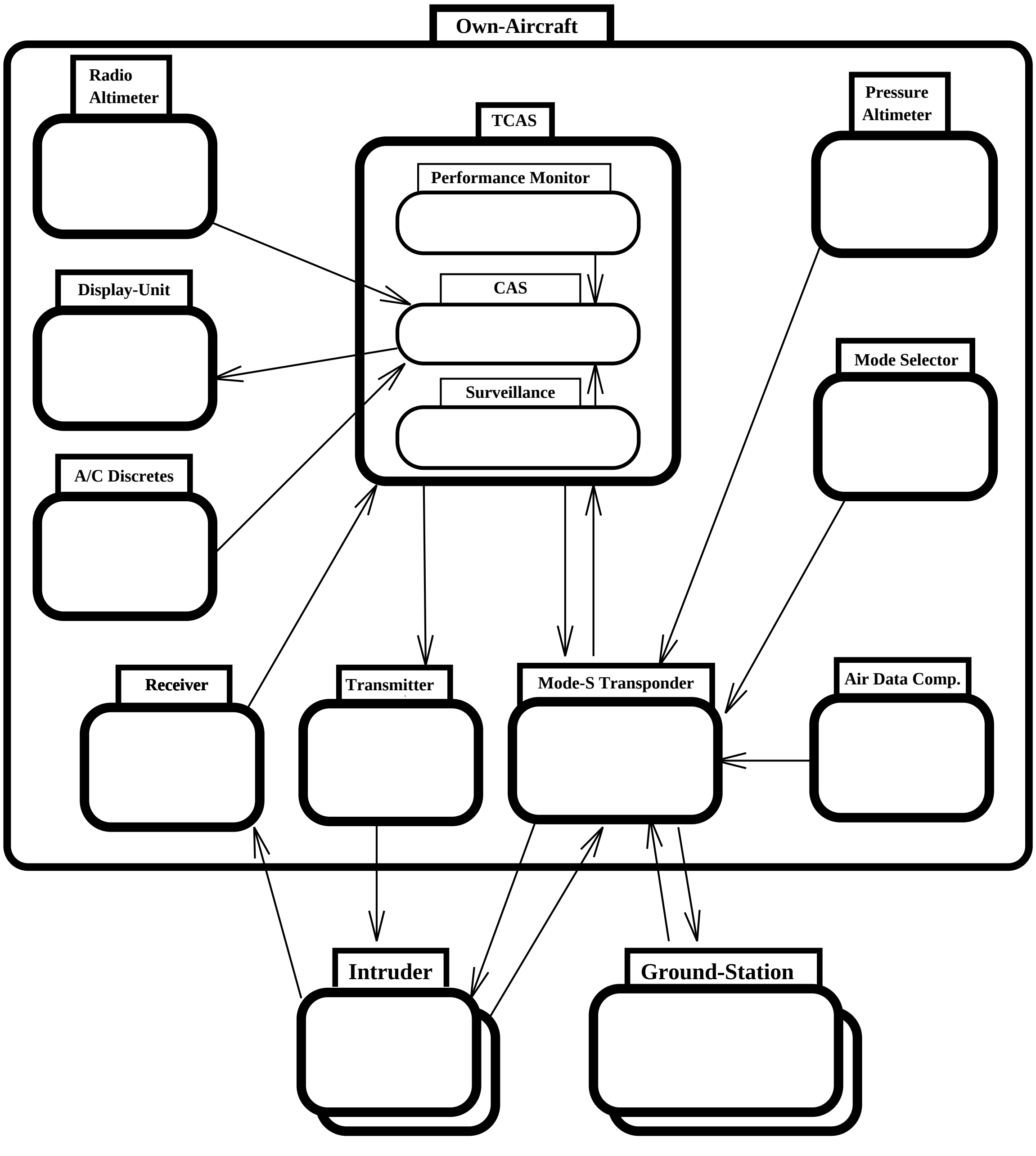


Figure 20: Component Communication

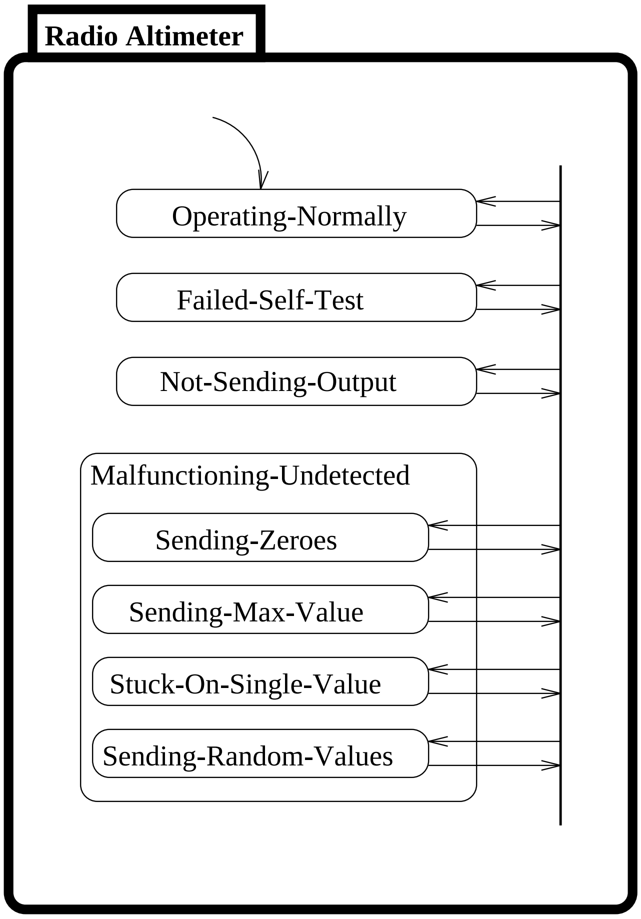


Figure 21: Altimeter component

Interface:

Source: Mode-S-Transponder
Destination: CAS

Trigger Event: RECEIVE(Sensitivity-Level-Command(IIS, SLC))
Condition:

		OR	
AND	$2 \leq \text{SLC} \leq 7$	T	·
	SLC = 15	·	T

Assignment(s):

$$\text{Mode-S-Ground-Station[IIS]} \triangleright \text{ Ground-Commanded-SL}_{\text{v-294}} = \begin{cases} \text{Cancel} & \text{if SLC} = 15 \\ \text{SLC} & \text{if } 2 \leq \text{SLC} \leq 7 \end{cases}$$

Output Action: None
Description:

If a sensitivity level command is received from own transponder, then set Ground-Commanded-SL of the appropriate ground station parallel state.

MOPS Ref.: SL_command_processing (p. 3-P21)

Figure 22: An example of an input interface definition

Interface:

Source: CAS
Destination: Mode-S-Transponder

Trigger Event: Received-Intruder-Intent-Event$_{\text{e-381}}$
Condition: true

Assignment(s):

VRC = Encode-RAC$_{\text{f-336}}$(Vertical-RAC$_{\text{v-154}}$,
Horizontal-RAC$_{\text{v-255}}$)
ARA = Encode-ARA$_{\text{f-335}}$(Climb-RA$_{\text{v-151}}$,
Descend-RA$_{\text{v-152}}$)

Output Action: SEND(Coordination-Update(VRC, ARA))

Description: This sends a coordination update message to own transponder.

MOPS Ref.: RESOLUTION_MESSAGE_PROCESSING (p. 3-P11).

Comments: ARINC 735 specifies the format of the coordination update message. It contains additional fields, such as sensitivity level, that are not specified in the pseudocode.

Figure 23: An example of an output interface definition

Name: `Mode S All-Call Reply (squitter)`
Message Format: DF-11 (All-Call Reply)
MOPS Reference: Detection 2.2.8.2.1

Source: Mode S equipped aircraft.
Destination: Broadcast.
Timetype: S-R or Periodic (as squitters at maximum period of 1.2s)

Data Representation:

DF Downlink Format 1 5	CA=0 Transponder Capability 6 8	AA Address Announced 9 32	PI Parity/-Identity 33 56

Contained Subfields:

Field	**Description**
DF	Defines the type of transmission. DF transmissions are replies. 11 = All-Call Reply
CA	Reports the capability of the transponder. The codes are: 0 = No extended capability report available. 1 = Comm A/B and extended capability report available. 2 = Comm A/B/C and extended capability report available. 3 = Comm A/B/C/D and extended capability report available. 4-7 = Not assigned
AA	Contains aircraft address in the clear.
PI	Ref.B 4.1.

Comments:

Generated as a reply to ground sensor all-call interrogation or as squitters.

Figure 24: Message format definition

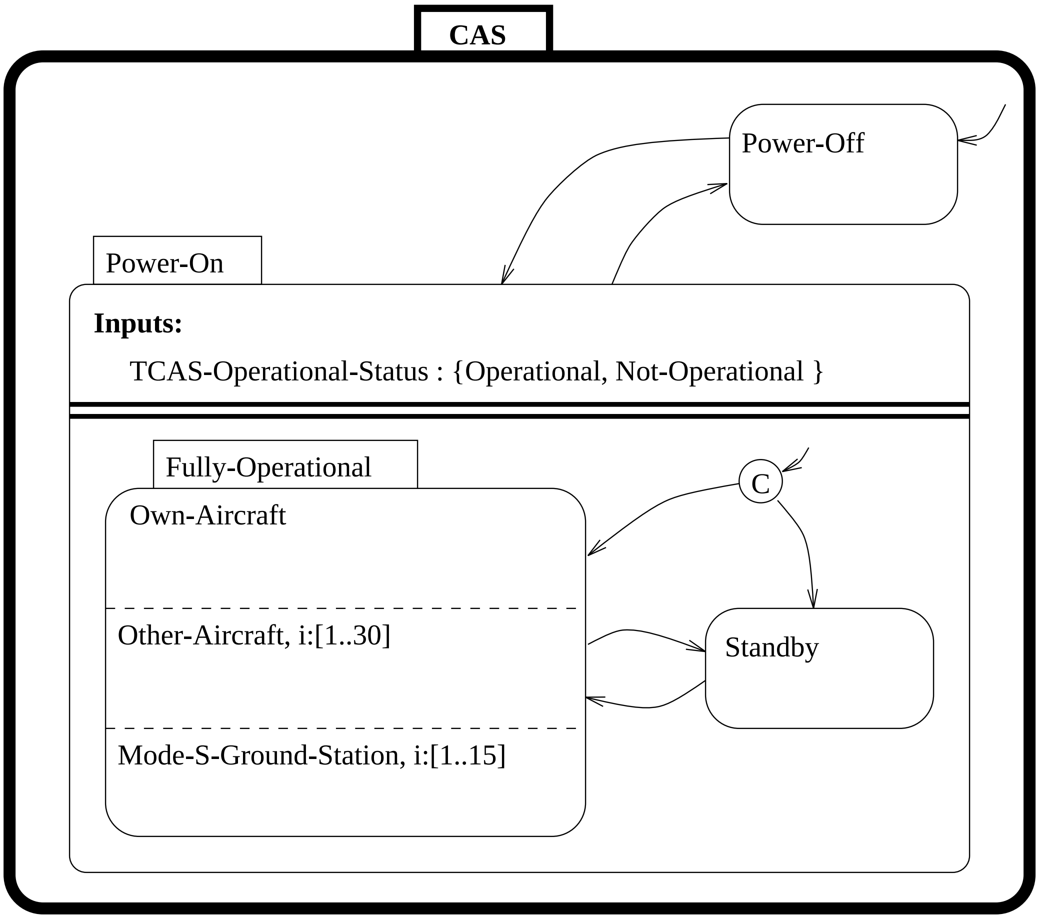


Figure 25: Collision Avoidance System

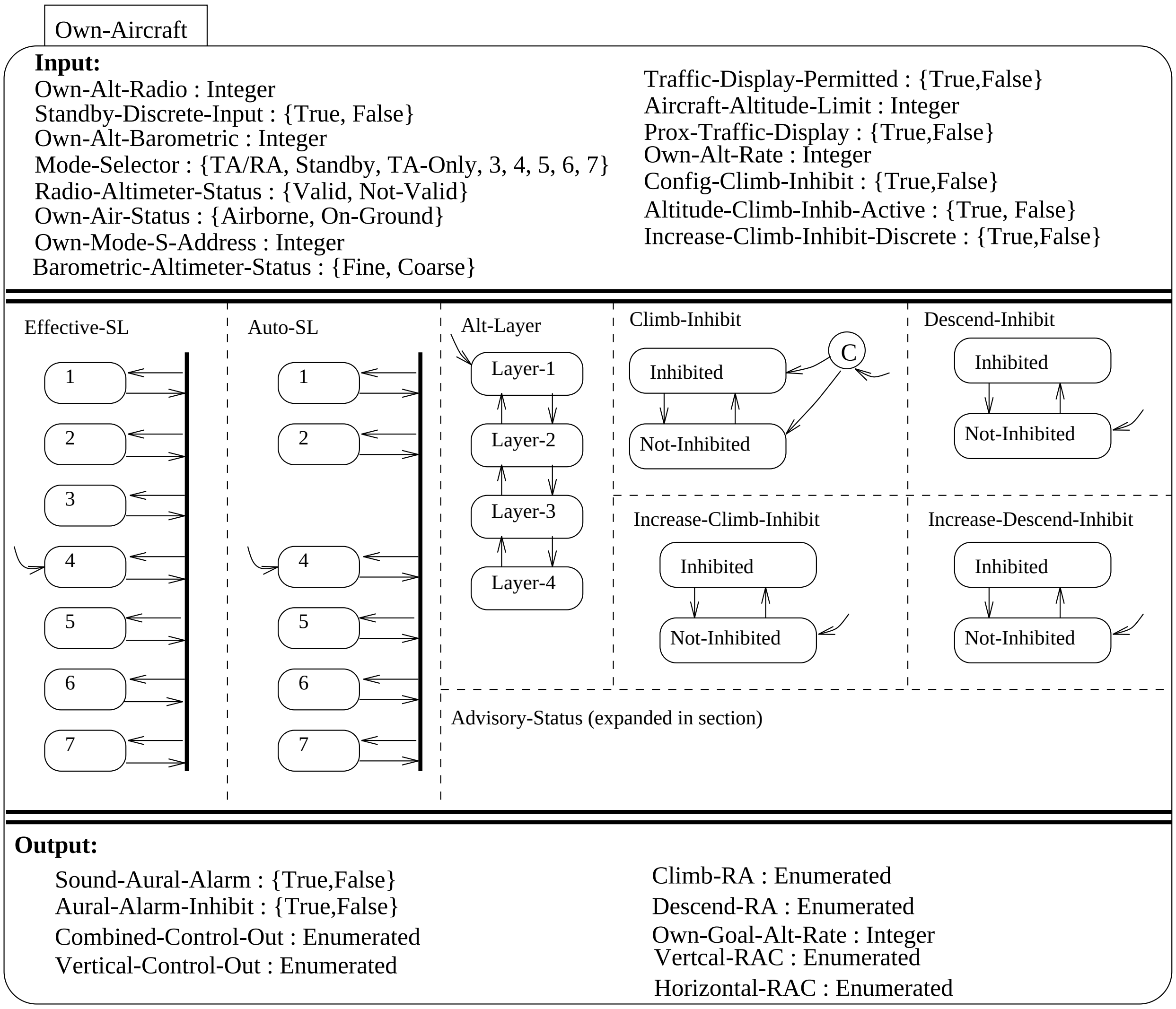


Figure 26: Own-Aircraft

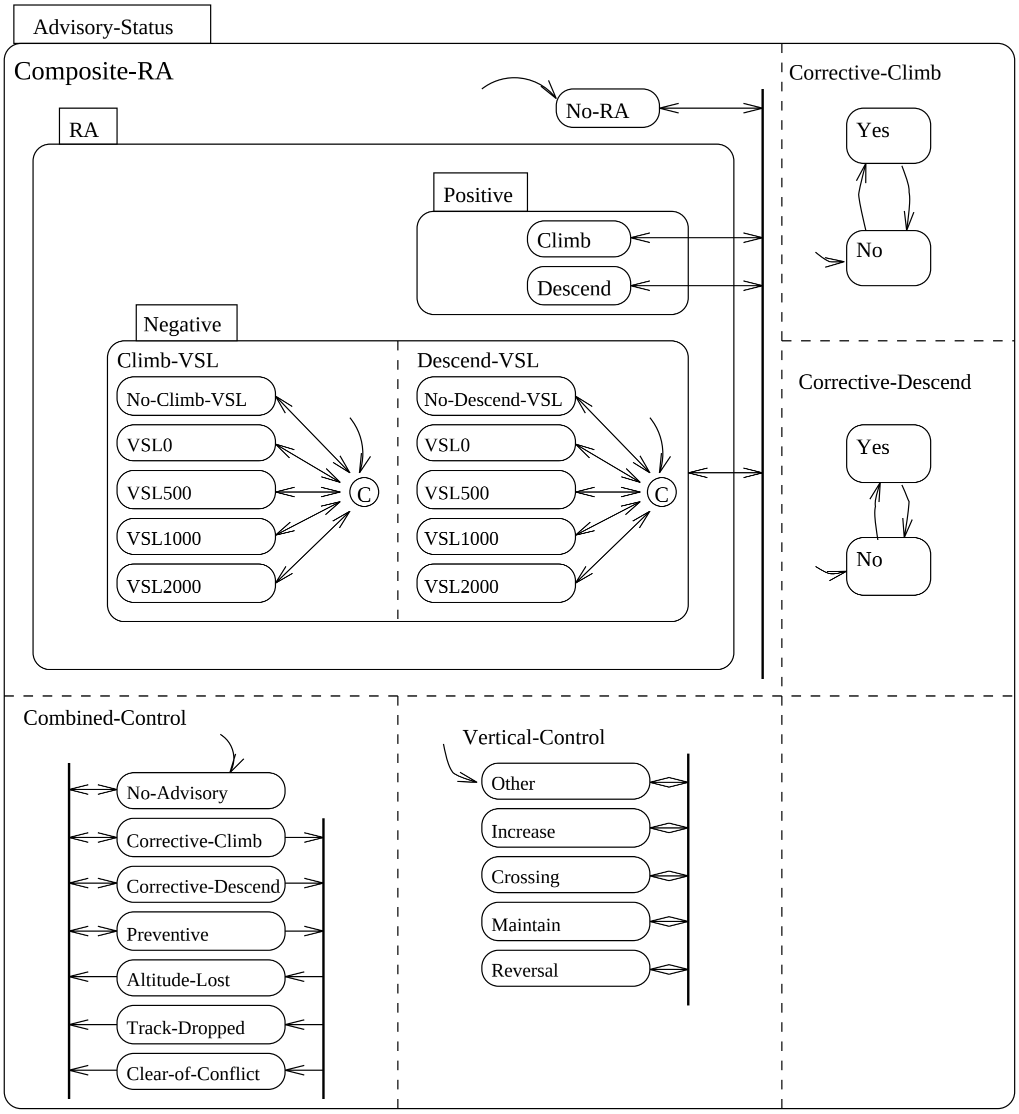


Figure 27: Advisory status

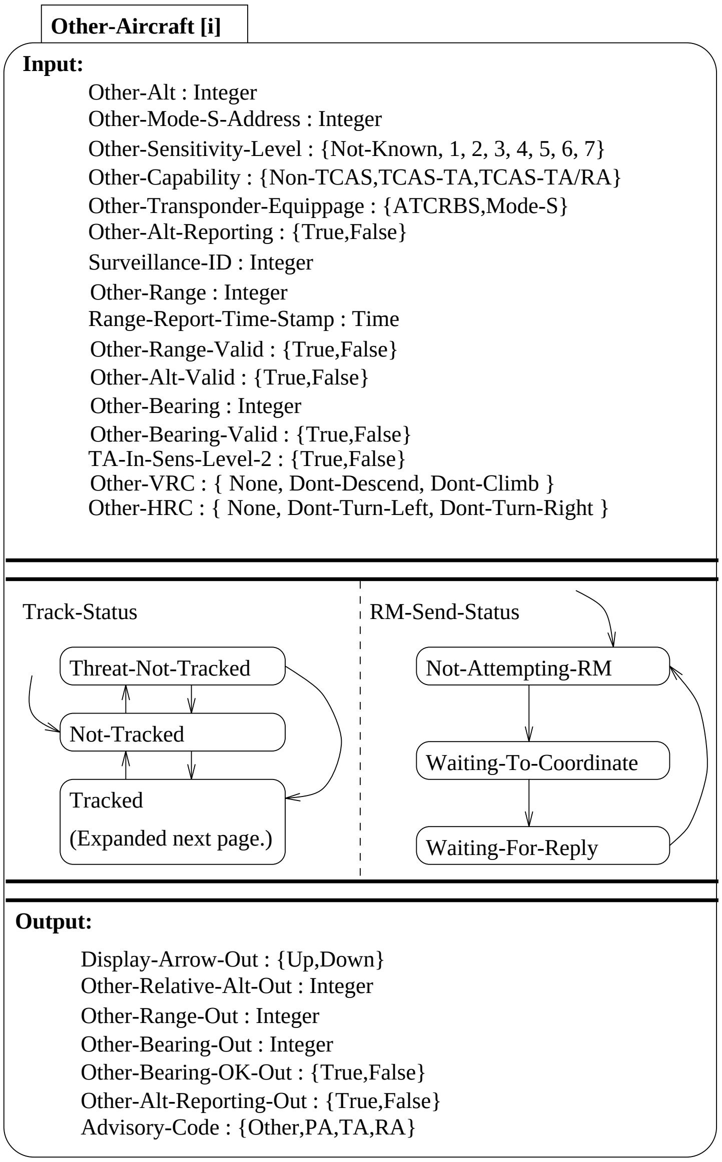


Figure 28: Other Aircraft Overview

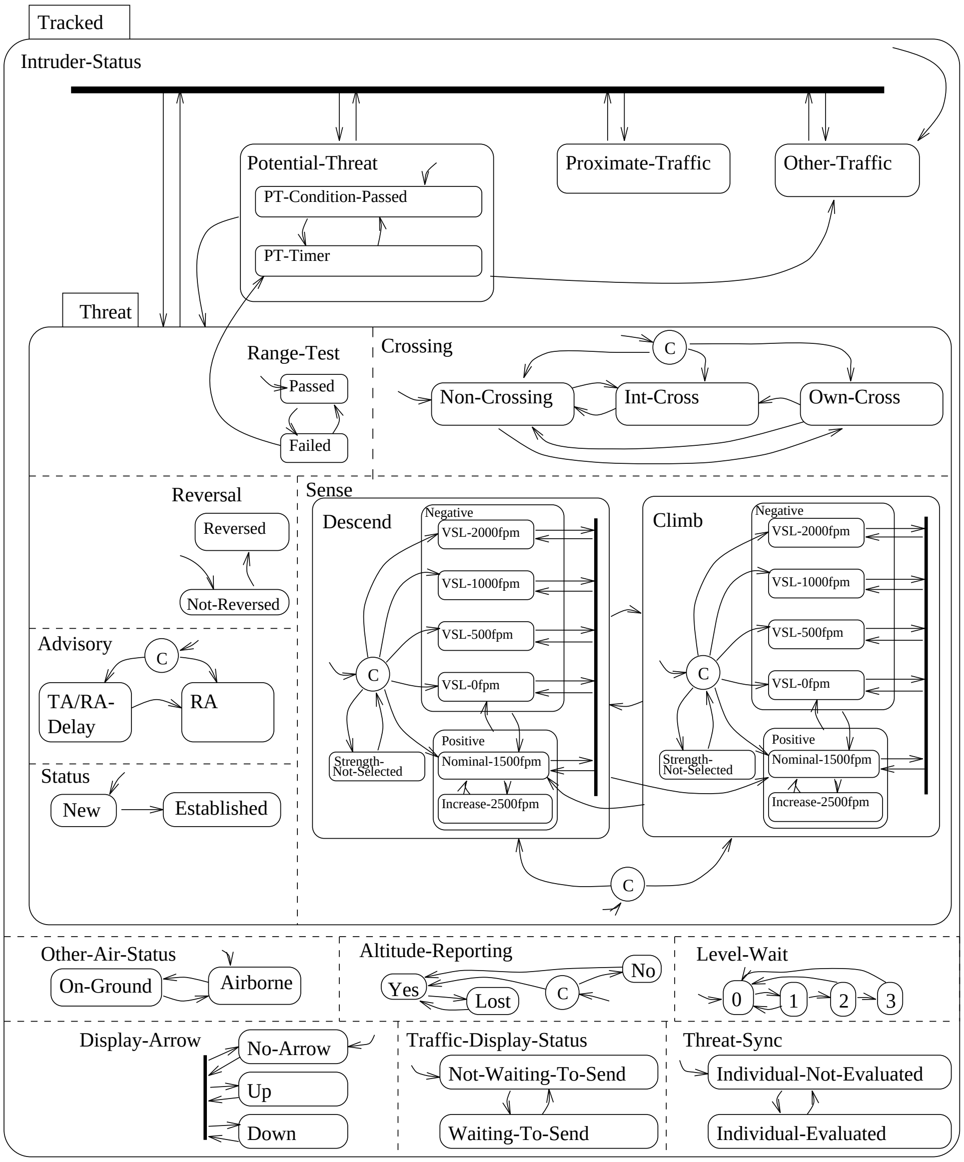


Figure 29: Tracked

Input: Other-Mode-S-Address

Location: Other-Aircraft$_{s\text{-}202}$

Source: Surveillance

Type: Integer

Expected Range: 1 ... $(2^{24} - 2)$

Granularity: 1 (unit)

Units: N/A

Load: N/A

Exception handling information: Mode-S addresses outside the valid range (i.e., all 0s or all 1s) are presently ignored. It is assumed that no such addresses will occur because administrative procedures will preclude this event. Duplicate addresses are treated similarly.

MOPS Reference: IDINTR

Description: The unique address of the Other-Aircraft.

Comments: This field has no meaning for non-Mode-S-equipped aircraft. No decision has been made about what to do about addresses that are outside the valid range or duplicate addresses due to failures of administrative procedures.

Figure 30: Input variable definition

Output: Display-Arrow-Out

Location: Other-Aircraft$_{\text{s-202}}$

Destination: Display-Unit

Type: Enumerated

Expected Range: {No-Arrow, Up, Down}

Granularity: N/A

Trigger: Display-Arrow-Evaluated-Event$_{\text{e-381}}$

Value:

Value	Condition
No-Arrow	Display-Arrow **in state** No-Arrow
Up	Display-Arrow **in state** Up
Down	Display-Arrow **in state** Down

Units: N/A

Load: 1/s for CAS.
MOPS Reference: ARROW

Description: From ATA-STD-TCAS II/1A 4.2.1.10: "A vertical arrow shall be placed to the immediate right of the traffic symbol if the vertical speed of the intruder is equal to or greater than 500 fpm with the arrow pointing up for climbing traffic and down for descending traffic."

Comments:

Figure 31: Output variable definition

Transition(s): $\boxed{\text{Threat}} \longrightarrow \boxed{\text{Other-Traffic}}$

Location: Other-Aircraft ▹ Tracked ▹ Intruder-Status$_{\text{s-237}}$

Trigger Event: Air-Status-Evaluated-Event$_{\text{e-379}}$

Condition:

AND		OR			
	Alt-Reporting$_{\text{s-202}}$ **in state** Lost	T	T	T	·
	Bearing-Valid$_{\text{m-298}}$	F	·	T	·
	Other-Range-Valid$_{\text{v-218}}$ = True	·	F	T	·
	Proximate-Traffic-Condition$_{\text{m-317}}$	·	·	F	·
	Potential-Threat-Condition$_{\text{m-314}}$	·	·	F	·
	Other-Air-Status$_{\text{s-202}}$ **in state** On-Ground	·	·	·	T

Output Action: Intruder-Status-Evaluated-Event$_{\text{e-379}}$

Description:

Columns 1-2 Lost altitude reporting and either the bearing or range inputs are invalid.

Column 3 Lost altitude reporting and both range and bearing are valid, but neither the proximate nor potential threat classification criteria are satisfied.

Column 4 Aircraft is on ground.

MOPS Ref. Section 7.1. TRAFFIC_ADVISORY.

Figure 32: Transition definition

Macro: Potential-Threat-Condition

Definition:

		OR	
A	Other-Air-Status$_{\text{s-202}}$ **in state** Airborne	·	T
N	Potential-Threat-Range-Test$_{\text{m-315}}$	T	T
D	Other-Alt-Reporting$_{\text{v-214}}$ = True	F	T
	Own-Tracked-Alt$_{\text{f-349}}$ $\geq$ 15500 ft$_{\text{(ABOVNMC)}}$	F	·
	Potential-Threat-Alt-Test$_{\text{m-313}}$	·	T

Description: To be considered a *Potential-Threat*, the intruder must satisfy the potential threat range criteria. If the intruder is altitude reporting, it must also satisfy the potential threat altitude criteria. If the intruder is not altitude reporting, then it is considered a potential threat only if own altitude is below 15500 ft$_{\text{(ABOVNMC)}}$.

MOPS Ref. TRAFFIC_ADVISORY.Traffic_advisory_detection, Range_hit_processing.

Figure 33: Macro definition

Function: Vertical-Resolution-Complement(i)
Return type: { 0, 1, 2 }
Definition:

Vertical-Resolution-Complement =

$$\begin{cases} 0 & \textbf{if} \ \text{Other-Aircraft}_{\text{s-202}}[\text{i}] \ \textbf{not in state} \ \text{Threat} \\ 1 & \textbf{if} \ \text{Other-Aircraft}_{\text{s-202}}[\text{i}] \ \textbf{in state} \ \text{Threat} \triangleright \ \text{Descend} \\ 2 & \textbf{if} \ \text{Other-Aircraft}_{\text{s-202}}[\text{i}] \ \textbf{in state} \ \text{Threat} \triangleright \ \text{Climb} \end{cases}$$

Description: This function returns the value of the Vertical-Resolution-Complement Mode S message field. Its values have the following meaning:

Value	Meaning
0	No vertical resolution advisory complement sent.
1	Don't descend.
2	Don't climb.

Explanation of value selection criteria: If Other-Aircraft is not in state Threat, then Vertical-Resolution-Complement has value 0 (no vertical RA complement). If TCAS has selected a Descend sense RA against the intruder, then Vertical-Resolution-Complement is set to 1 (don't descend). Likewise, if TCAS has selected a Climb sense RA against the intruder, then Vertical-Resolution-Complement is set to 2 (don't climb).

MOPS Ref.: Send_initial_intent (p. 6-P57).

Figure 34: Function definition

References

[BGFG86] G. R. Bruns, S. L. Gerhart, I. Forman, and M. Graf. Design technology assessment: The statecharts approach. Technical Report STP-107-86, MCC, March 1986.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[FBWJ92] S. Faulk, J. Brackett, P. Ward, and J. Kirby Jr. The core method for real-time requirements. *IEEE Software*, 9(5), September 1992.

[FG79] M. Fitter and T. R. G. Green. When do diagrams make good computer languages. *International Journal on Man-Machine Studies*, 11, 1979.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hei94] Mats P.E. Heimdahl. *Static Analysis of State-Based Requirements: Analysis for Completeness and Consistency.* PhD thesis, University of California, Irvine, 1994.

[Hen80] K. L. Heninger. Specifying software for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol91] G. J. Holzmann. *Design and validation of computer protocols.* Prentice Hall, 1991.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[HP87] D. Hatley and I. Pirbhai. *Strategies for Real Time System Specification.* Dorset House Publishing, 1987.

[Jaf88] M.S. Jaffe. *Completeness, Robustness, and Safety in Real-Time Software Requirements and Specifications.* PhD thesis, University of California, Irvine, 1988.

[JLHM91] M. S. Jaffe, N. G. Leveson, M. P.E. Heimdahl, and B. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[LCS91] N.G. Leveson, S.S. Cha, and T.J. Shimeall. Safety verification of ada programs using software fault trees. *IEEE Software*, July 1991.

[LH83] N.G. Leveson and P.R. Harvey. Analyzing Software Safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[LS87] N. G. Leveson and J. L. Stolzy. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, 13(3):386–397, March 1987.

[Mel91] B. E. Melhart. An external interaction model for specifying requirements of embedded software. Technical Report Draft, Texas Christian University, Jan 1991.

[Par79] D.L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[PS89] A. Pnueli and M. Shalev. What is in a step? In J. Klop, J. Meijer, and J. Rutten, editors, *J.W. De Baker, Liber Amicorum*, pages 373–400. CWI Amsterdam, 1989.

[PW89] D. L. Parnas and Y. Wang. The trace assertion method of module interface specification. Technical Report 89-261, Queen's University, Kingston, Ontario K7L3N6, 1989.

[RR91] A. P. Ravn and H. Richel. Requirements capture for embedded real-time systems. In *IMACS Symposium MCTS*, 1991.

[Sha92] A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9), September 1992.

[vS90] A. J. van Schouwen. The A-7 requirements model: Re-examination for real-time sysems and an application to monitoring systems. Technical Report 90-276, Queen's University, Kingston, Ontario, May 1990.

[WGS94] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from bolean specification. *IEEE Transactions on Software Engineering*, 20(5), May 1994.

[WM85] P. Ward and S. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, 1985.