Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future

Nancy G. Leveson[1], Mats P.E. Heimdahl[2], and Jon Damon Reese

[1] Aeronautics and Astronautics Dept.
MIT
Room 33-406, 77 Massachusetts Ave.
Cambridge, MA 02139-4307
leveson@mit.edu

[2] Computer Science and Engineering Department,
University of Minnesota,
4-192 EE/CS Building, 200 Union Street S.E.
Minneapolis, MN 55455,
heimdahl@cs.umn.edu

Abstract. Previously, we defined a blackbox formal system modeling language called RSML (Requirements State Machine Language). The language was developed over several years while specifying the system requirements for a collision avoidance system for commercial passenger aircraft. During the language development, we received continual feedback and evaluation by FAA employees and industry representatives, which helped us to produce a specification language that is easily learned and used by application experts.

Since the completion of the RSML project, we have continued our research on specification languages. This research is part of a larger effort to investigate the more general problem of providing tools to assist in developing embedded systems. Our latest experimental toolset is called SpecTRM (Specification Tools and Requirements Methodology), and the formal specification language is SpecTRM-RL (SpecTRM Requirements Language).

This paper describes what we have learned from our use of RSML and how those lessons were applied to the design of SpecTRM-RL. We discuss our goals for SpecTRM-RL and the design features that support each of these goals.

1 Introduction

In 1994, we published a paper describing a blackbox formal system modeling language called RSML (Requirements State Machine Language). The language was developed over several years during an effort to specify the system requirements for a collision avoidance system for commercial passenger aircraft called TCAS II (Traffic Alert and Collision Avoidance System). Because this was to be the official FAA (Federal Aviation Administration) specification, it was developed with continual feedback and evaluation by FAA employees, airframe manufacturers, airline representatives, pilots, and other external reviewers. Most of the reviewers were not software engineers or even computer scientists, and we believe this helped in producing a specification language that is easily learned and used by application experts. RSML is still being used by the FAA, its subcontractors, and RTCA committees to specify the upgrades and changes to TCAS II.

Those designing specification languages often have themselves in mind as potential users. However, our familiarity with certain notations, especially mathematical notations, such as predicate calculus, hides their weaknesses. Our first attempts at designing RSML, therefore, were failures: Our notation was clear to us but not to the representatives from the airframe manufacturers, component subcontractors, airlines, and pilot groups that reviewed the TCAS specification during its development. The feedback from a diverse group of users helped us to evaluate the evolving specification language more objectively in terms of what did and did not need to be in the language; how to satisfy our language design criteria; and its practicality, feasibility, and usability.

Due to pressure to meet FAA deadlines for getting TCAS II on aircraft, we were unable to use immediately all the lessons learned from that experience and apply it to the design of RSML. Since that time, we have specified several additional systems including a robot, flight management system, and air traffic control components, each time learning more lessons about the design of formal specification languages. Our research goal is to determine how specification languages can be designed to reflect these lessons. Our research paradigm is to determine important goals for specification languages from experience with industrial applications, to generate hypotheses about how these goals might be accomplished, and then to instantiate these hypotheses in the design of a specification language that we will use in future experimentation. In this way, we hope to build knowledge incrementally about how to most effectively design specification languages.

Our specification language research is part of a larger research effort to investigate the more general problem of providing tools to assist in developing embedded systems. Our latest experimental toolset is called SpecTRM (Specification Tools and Requirements Methodology), and the formal specification language is SpecTRM-RL (SpecTRM Requirements Language). In addition to the general goals we had for designing RSML [9], the lessons we have learned to date have focused our latest efforts on solving the following problems:

1. Through the use of RSML, we have determined that readability and reviewability by domain experts can be further enhanced by minimizing the semantic distance between the reviewer's mental model and the constructed models. The problem we are now addressing is how to construct a modeling language that will allow and encourage modelers to reduce this semantic distance in the models they build.
2. Specifiers are used to including internal design in their specifications and seem to have difficulty building pure blackbox requirements models. So a

second goal was to provide more support and guidance in building software requirements (versus software design) models.
3. We found certain common features of formal specification languages were very error-prone in use. In particular, the use of internal broadcast events accounted for most of the errors found by reviewers of the TCAS specification and also contributed substantially to the difficulty reviewers had in reading the models. A third goal for SpecTRM-RL was to determine if such internal events can be effectively eliminated from state-based modeling languages.
4. Formal models are expensive to produce. Thus, reuse of at least parts of the language should be supported by the language design. Such features should also support the design of models for product families.
5. Accidents and major losses involving computers are usually the result of incompleteness or other flaws in the software requirements, not coding errors [8, 12]. We previously defined a set of formal criteria to identify missing, incorrect, and ambiguous requirements for process-control systems [6, 8]. Engineers have made the criteria into checklists and used them on a variety of applications, such as radar systems, the Japanese module of the Space Station, and review criteria for FDA medical device inspectors. Two goals for SpecTRM-RL are to determine (1) how to enforce as many of the constraints as possible in the syntax of the language and (2) how to design the language to enhance the ability to manually check or build tools to automatically check the specifications for the criteria that cannot be enforced by the language design itself.

This paper describes what we have learned from our experimentation with the design of SpecTRM-RL about achieving the first four goals. Our results for the fifth goal will be described in a future paper. The design features of SpecTRM-RL that support each of these goals are described but a complete description of the language, including its syntax, is beyond the scope of this paper. We are currently producing a SpecTRM-RL language design manual and automated tools to assist in experimental use of the language.

2 Enhancing Usability and Reviewability

The primary goal for the design of a specification language should be to make the representation appropriate for the tasks to be performed by the users, i.e., to make the design *user-centered* (rather than designed primarily to make analysis easier or to be faithful to standard mathematical conventions). Software is a human product and specification languages are used to help humans perform the various problem-solving activities involved in software engineering. Our goal is to provide specifications that support human problem-solving and the tasks that humans must perform in software development and evolution as well as to allow automated analysis. We attempt to support human problem-solving by grounding specification design on psychological principles of how humans use specifications to solve problems as well as on basic system engineering principles.

We discuss two of these aspects here: minimizing semantic distance (problem 1 above) and building blackbox specifications (problem 2 above). Problems 3 and 4, as they reflect on the design of SpecTRM-RL, are discussed in later sections of this paper.

An important psychological principle for enhancing reviewability is the concept of semantic distance [5]. We define an informal concept of *semantic distance*, similar to Norman's use of the term, 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 a requirements specification, the semantic distance between their understanding of the required 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 components and state variables of the controlled system. Specifically, "private" variables and procedures (functions) 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. That is, the specification should be black box.

A blackbox model of behavior permits statements and observations to be made only in terms of outputs and the inputs that stimulate or trigger those outputs. The model does not include any information about the internal design of the component itself, only its externally visible behavior. The overall system behavior is described by the combined behavior of the components, and the system design is modeled in terms of these component behavior models and the interactions and interfaces among the components.

When the description of the required controller behavior includes more than just its blackbox behavior (e.g., it includes software design information), then the semantic distance between the required process-control behavior and the specified controller behavior increases and the relationship between them becomes more difficult to validate (d_1 vs. d_4 in Figure 1). In fact, if adequately efficient code can be generated from the requirements specification directly, then an internal design specification may never be needed. "Adequately efficient" must be determined for each specific application's timing requirements. We are working on this code-generation problem [7].

In addition, 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. We discuss this further in the next section.

Blackbox requirements specification languages not only enhance readability and reviewability, but they also simplify the transition from system requirements and system design to software requirements. The gap between system design and software requirements is frequently cited as a major problem in our interactions with industry. We believe some of the problem stems from the fact that software requirements often contain a lot of software design decisions, which makes the gap between the two specifications larger and more complex to negotiate.

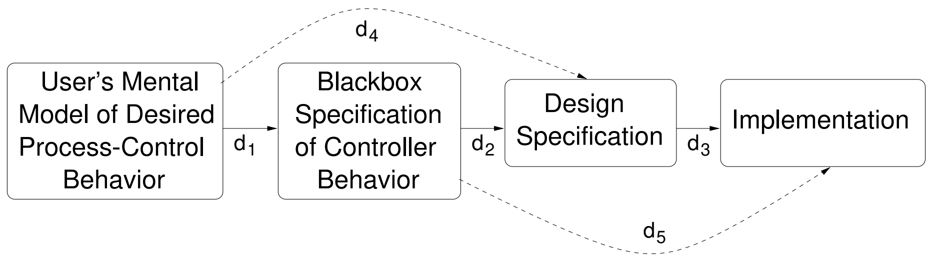


Fig. 1. Reviewability increases as the semantic distance between the user's mental model of the desired behavior and the specification (d_1 vs. d_4) decreases.

2.1 Minimizing Semantic Distance

Our language is designed primarily for process-control systems. Therefore we attempt to minimize the semantic distance d_1 by basing the specification language design on fundamental process control principles. In process control, 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 2). This system function consists of the functional description and the set of constraints on the system [11]. At any moment, there is a unique set of relationships between inputs and outputs whereby each output value will be related to the past and present values of the inputs and time. These relationships will involve fundamental chemical, thermal, mechanical, aerodynamic, or other laws as embodied within the nature and construction of the system. The system is constructed from components whose interaction implements F including, usually, a controller or controllers whose function is to ensure that F is correctly achieved.

A typical process-control system can be divided into four types of components: the process, sensors, actuators, and controller (see Figure 2).

Process: 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 $\mathcal{V}_m \times \mathcal{I}_s \times \mathcal{D} \times t \rightarrow \mathcal{O}_s \times \mathcal{V}_c$.

Sensors: These devices 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: These 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 $\mathcal{O} \times t \rightarrow \mathcal{V}_m$.

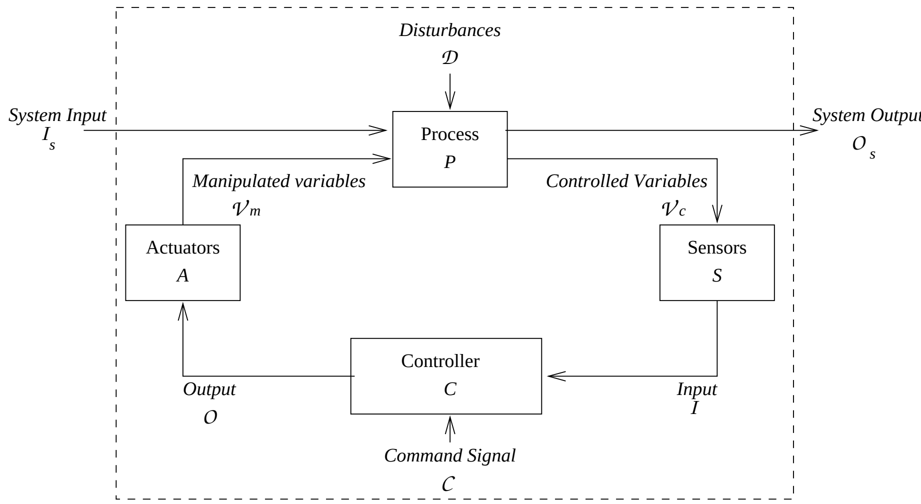


Fig. 2. Basic Process Control Loop

Controller: 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. *Feedback* is provided via the controlled variables in order to monitor the behavior of the process. This feedback information (along with external command signals $\mathcal{C}$) can be used as a foundation for future control decisions as well as an indicator of whether the changes in the process initiated by the controller have been achieved.

To reason about this type of process-controlled system, Parnas and Madey defined what they call the four-variable model [14]. This model is essentially an abstraction of part of the traditional feedback process control model presented here. The approach to modeling used in Parnas Tables [13] and SCR [4, 3] are based on this four variable model and, thus, built upon this same classic process control model.

The model presented in this section 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 a 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 such a system 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.

All of this information can be embedded in a state-machine model of the controlled process, and we specify the blackbox behavior of the controller (i.e., the function F_C to be computed by the controller) using such a state machine model. In SpecTRM-RL models, 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[1], cockpit displays), 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 and transponders).

The state machine model of the control function must be iteratively fine tuned during requirements development to mimic the current understanding of the real-world process and the required controller behavior. The state machine is essentially an abstraction of the behavior of the system function because 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.

3 Building Blackbox Specifications in SpecTRM-RL

Although RSML allows blackbox behavior specifications, the language itself does not encourage or enforce them. We found people tend to include design in the specification when using general state-machine modeling languages such as RSML or Statecharts. SpecTRM-RL, therefore, was not designed to be a general modeling language, but rather specifically designed to create blackbox

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

requirements specifications to define an input/output process-control function, as is also true of SCR and Parnas Tables. General modeling features not needed for blackbox specifications are not included in SpecTRM-RL, and new abstractions (such as modes) are included that assist in blackbox modeling of control system components. Thus, SpecTRM-RL is not just another variant of Statecharts although there are some superficial similarities. Like SCR and Parnas Tables, SpecTRM-RL enforces the specification of an input/output process control function. Statecharts allows much more general models to be built.

In order to make our language formal enough to be analyzable (and yet readable and reviewable by non-mathematicians), we have defined a formal model (RSM or Requirements State Machine) that underlies a more readable specification language or languages. The RSM is a general behavioral model of the required control function with the components of the state machine mapped to the appropriate components of the control loop. This model has been published previously [6], and we do not refer to it further in this paper. We note only that the underlying model is a Mealy automaton, as is the model for SCR, Parnas Tables, Statecharts, and most other languages based on state-machines.

The higher-level specification language based on this underlying model must allow the modeler to specify the required process-control function F_C. Figure 3 shows a more detailed view of the components of the control loop, including distinguishing between human and automated controllers.

All control software (and any controller in general) uses an internal model of the general behavior and current state of the process that it is controlling. This internal model may range from a very simple model including only a few variables to a much more complex model with a large number of state variables and transitions. The model may be embedded in the control logic of an automated controller or in the mental model maintained by a human controller. It is used to determine what control actions are needed. The model is updated and kept consistent with the actual system state through various forms of feedback.

The design of SpecTRM-RL is influenced by our desire to perform safety analysis on the models. When the controller's model of the system diverges from the actual system state, erroneous control commands (based on the incorrect model) can lead to an accident—for example, the software does not know that the plane is on the ground and raises the landing gear or it does not identify an object as friendly and shoots a missile at it. The situation becomes more complicated when there are multiple controllers (both human and automated) because their internal system models must also be kept consistent. In addition, human controllers interacting with automated controllers must also have a model of the automated controllers' behavior in order to monitor or supervise the automation as well as the controlled system itself.

One reason the models may diverge is that 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 po-

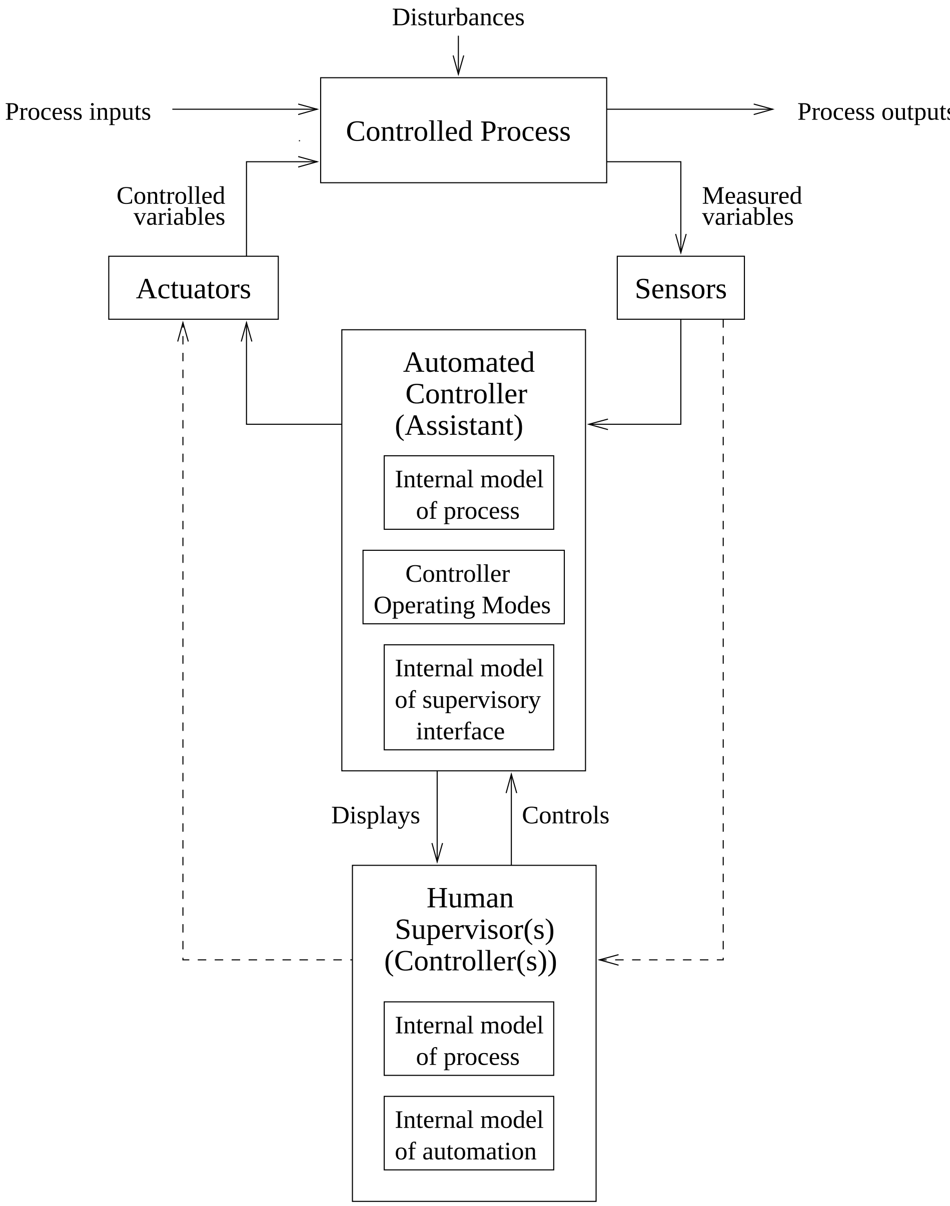


Fig. 3. A basic control loop. A blackbox requirements specification captures the controller's internal model of the process. Accidents occur when the internal model does not accurately reflect the state of the controlled process.

sitions). However, at any time, the controller has only measured values of the component states (which may be subject to time lags 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 controlled process (the aircraft trajectory) may be caused by aircraft performance limitations.

The automated controller also has a model of its interface to the human controllers or its supervisor(s). This interface, which contains the controls, displays, alarm annunciators, etc. is important because it is the means by which the two controllers' models are synchronized. Each of these components is included explicitly in our models and modeling language. We represent the controlled process and supervisory interface models using state machines and define required behavior in terms of transitions in this machine. The controller outputs (commands to the actuators) are specified with respect to state changes in the model as information is received about the current state of the controlled process via controlled variables read by sensors.

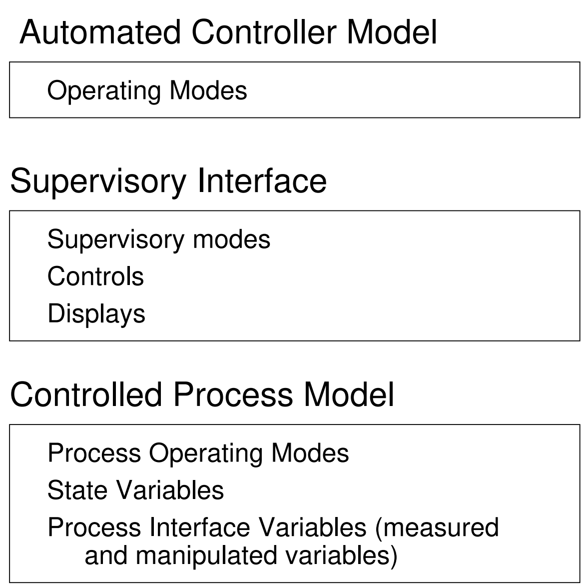


Fig. 4. A SpecTRM-RL model has three parts.

Thus a SpecTRM-RL specification of control software is composed of three interrelated models (Figure 4): (1) a specification of the operating modes of the controller, (2) a specification of the controllers's view of its supervisory interface (the component or components, including human operators, that are controlling it), and (3) a model of the controlled process.

3.1 The Structure of a SpecTRM-RL Specification

Engineers often use modes in describing required system functionality. Mode confusion also is frequently implicated in the analysis of operator errors that lead to accidents. We have included in SpecTRM-RL the ability to describe behavior in terms of modes both to reduce semantic distance (and enhance reviewability) and to allow for analysis of various types of mode-related errors [10].

A mode can be defined as a mutually exclusive set of system behaviors. For example, the following table shows the possible transitions between states in a simple state machine given two system modes: startup mode and normal operation mode.

	s_1	s_2	s_3	s_4	s_5
Startup mode	s_3	s_2	s_4	s_5	s_1
Normal mode	s_3	s_4	s_1	s_5	s_1

Table 1. A simple state machine with two modes defined using a standard state transition table. The states in the machine (listed at the top of the table) are s_1 through s_5 while the conditions under which the transition is made are listed on the left (e.g., startup mode and normal mode). Note that transitions may depend on more conditions than simply the current processing mode.

The startup and normal processing modes in this machine determine how the machine will behave over the entire set of state transitions. For example, if the conditions occur that trigger a transition from state s_3, the machine will transfer to state s_4 if it is in startup mode or to state s_1 if it is in normal processing mode. Note that modes are simply states that play a particular role in the state machine behavior (i.e., control a sequence or set of state transitions). That is, they are a convenient abstraction for describing and understanding complex system behavior, but they do not add any power to the state machine description. In general, state transitions may be triggered by events, conditions, or simply the passage of time. The current operating mode determines how these triggers will be interpreted and what transitions will be taken. Note that there is no real difference between a state and a mode by this definition. Any conditions or states could be labelled a "mode" (which indeed is done in some specification languages), although this is not very helpful and is not the way engineers use the term "mode".

Modern aircraft and other complex systems often have a large number of operating modes and possible combinations of operating modes. In the modeling and analysis of control systems, we find it useful to distinguish between three different types of modes:

1. *Supervisory modes* determine who or what is controlling the component at any time. Control loops may be organized hierarchically, with multiple controllers or components, each being controlled by the layer above and controlling the layer below. In addition, each component may have multiple controllers (supervisors). For example, a flight control computer in an aircraft may get inputs from the flight management computer or directly from the pilot. Required behavior may be different depending on what supervisory mode is currently in effect. Mode-awareness errors related to confusion in coordination between the multiple supervisors of a control component can be defined in terms of these supervisory modes.
2. *Component operating modes* control the behavior of the control component itself. They may be used to control the interpretation of the component's interfaces or to describe the component's required process-control behavior.
3. *Controlled-system* (or *plant* in control theory terminology) *operating modes* specify sets of related behaviors of the controlled system and are used to indicate its operational state. For example, an aircraft may be in takeoff, climb, level-flight, descent, or landing mode.

The use of modes does not violate the blackbox nature of SpecTRM-RL; they represent externally visible behavior (required functionality) of the component and not the internal design of the software to achieve that functionality. For example, *capture mode* (which can be *armed* or *not armed*) in the flight management system example shown in Figure 5 indicates whether the aircraft will automatically level off when a pilot-specified altitude constraint is reached. The pilot is responsible for setting the altitude constraint and (usually) for directly or indirectly selecting capture mode.

As stated earlier, a SpecTRM-RL specification has three interrelated models. The top box of Figure 5 shows the graphical part of an example specification of a flight management system. The system has seven modes of operation, all of which have only one value at any one time. The boxes shown under each mode label represents the discrete values for that mode, e.g., pitch can be in *altitude hold, vertical speed, indicated air speed*, or *altitude capture* mode. The line at the left of the choices simply groups the choices under the variable and indicates that only one may be active at any time and does not represent state transitions (as it did in RSML).

A second part of a SpecTRM-RL model is a specification of the component's view of its supervisory interface. The supervisory interface consists of a model of the operator controls and displays or other means of communication by which the component relays information to the supervisor. Note that the interface models are simply the view that the component has of the interfaces—the real state of the interface may be inconsistent with the assumed state due to various types of design flaws or failures. For example, a flight control computer in an aircraft may get inputs from the flight management computer or directly from the pilot. Required behavior may be different depending on what supervisory mode is currently in effect. By separating the assumed interface from the real interface, we are able to model and analyze the effects of various types of errors and

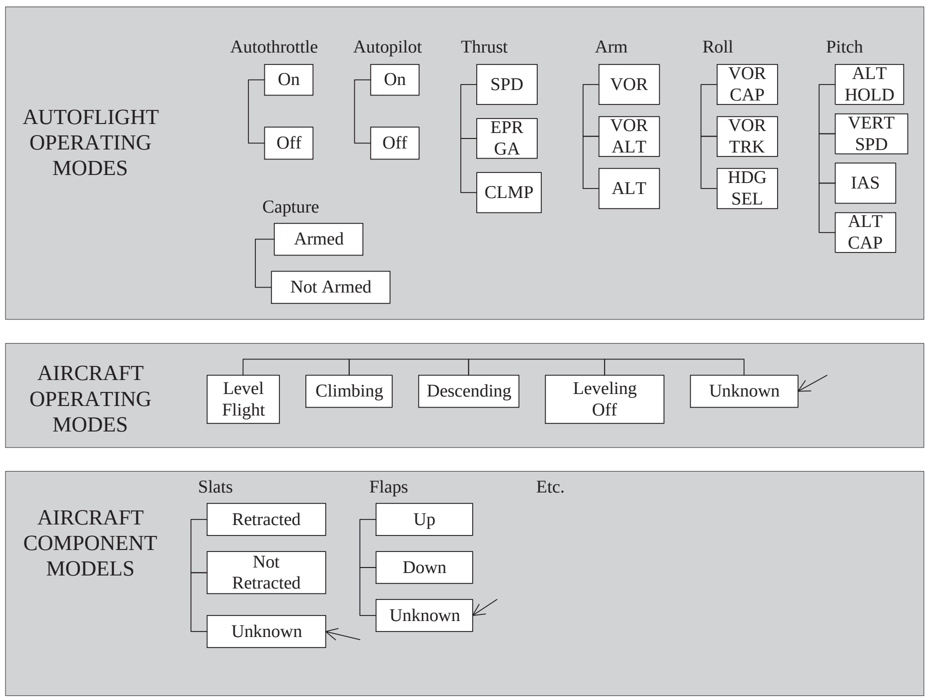


Fig. 5. Example of operating modes for a flight management system

failures (e.g., communication errors or display hardware failures). In addition, separating the physical design of the interface from the logical design (required content) will facilitate changes and allow parallel development of the software and the interface design.

The third part of a SpecTRM-RL model is the component's model of the controlled system (plant). The description of a simple component may include only a few relevant state variables. If the controlled process or component is complex, the model of the controlled process may itself be represented in terms of its operational modes and the states of its subcomponents. In a hierarchical control system, the controlled process may itself be a controller of another process. For example, the flight management system may be controlled by a pilot and may itself issue commands to a flight control computer, which issues commands to an engine controller. If, during the design process, components that already exist are used, then those plug-in component models could be inserted into the SpecTRM-RL process model.

If the SpecTRM-RL model is of a non-control component (e.g., a radar data processor), there might not be a supervisory interface. There will still be operating modes, however, and a model of the required input-output function to be computed by the component.

The language itself consists of a graphical model of the state machine, output message specifications, state variable definitions, operator interface variable definitions, state transition specifications, macros, and functions.

Graphical State Machine. The SpecTRM-RL notation is driven by the use of the language to define a function from outputs to inputs. SpecTRM-RL has a greatly simplified graphical representation (compared to RSML or Statecharts), which is made possible because we eliminated the types of state machine complexity necessarily for specifying component design but not necessary to specify the input/output function computed in a pure blackbox requirements specification. The architecture of the state transitions becomes so simple that we found no need to represent it in the graphical state machine—the transitions simply represent the changes between state variable values.

State values in square boxes represent *inferred values*. Inferred values are not input directly but represent the aspects of the process state model that must be inferred from measurements of monitored process variables. Inferred process states are used to control the computation of the control function. They are necessarily discrete in value[2], and thus can be represented as a finite state variable. In practice, such state variables almost always have only a few relevant values (e.g., altitude below 500 feet, altitude between 500 feet and 10,000 feet, altitude above 10,000 feet). State values denoted as circles or ovals represent direct input and output values (controlled or monitored variables).

[2] If they are not discrete, then they are not used in the control of the function computation but in the computation itself and can simply be represented in the specification by arithmetic expressions involving input variables.

The supervisory interface model shows the supervisory mode, which describes how this computer is being supervised, e.g., by a human, computer, or both (Figure 6). It also shows the state of the controls and the displays (including oral annunciations, etc.).

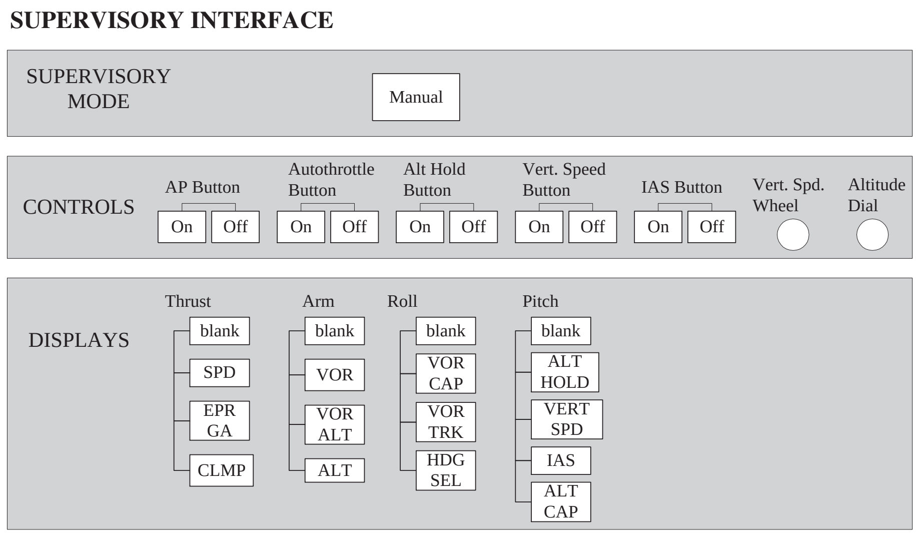


Fig. 6. Example of SpecTRM-RL model of the supervisory modes

Output Message Specification. Everything starts from outputs in SpecTRM-RL. By starting from the output specification, the specification reader can determine what inputs trigger that output and the relationship between the inputs and outputs. RSML did not explicitly show this relationship (although it could be determined, with some effort, by examining the specification). A simplified example is shown in Figure 7. More information is actually required by our completeness criteria than is shown in the example, for instance, specification of timing assumptions related to the message.

The conditions under which an output is triggered (sent) is simply a predicate logic statement over the various states, variables, and modes in the specification. During the TCAS project, we discovered that the triggering conditions required to accurately capture the requirements were often extremely complex. The propositional logic notation traditionally used to define these conditions did not scale well to complex expressions and quickly became unreadable (and error-prone). To overcome this problem, we decided to use a tabular representation of disjunctive normal form (DNF) that we call AND/OR tables. We have maintained this successful notation in SpecTRM-RL. The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated predicates. A dot denotes "don't care."

Output Message

Resolution Advisory

TRIGGERING CONDITION

Composite-RA$_{s\text{-}266}$ **in state** RA	T
Traffic-Display-Status$_{s\text{-}148}$ **in state** Waiting-To-Send	T

MESSAGE CONTENTS

FIELD	**VALUE**
Bits 11-17	Own-Goal-Altitude-Rate$_{v\text{-}219}$
Bits 18-20	Combined-Control$_{v\text{-}227}$
Bits 21-23	Vertical-Control$_{v\text{-}231}$
Bits 24-26	Climb-RA$_{v\text{-}233}$
Bits 27-29	Descent-RA$_{v\text{-}235}$

Fig. 7. Example of SpecTRM-RL output message specification

The subsecripts in the specification represent whether the value is a variable (v) or a state (s). The other alternatives, macros (m) and functions (f) are described later in this paper. The number attached to the subscript is the page on which the variable, state, macro, or function is defined.

State Variable Definition. State variable values come from inputs or they may be computed from such input values or inferred from other state variable values. Figure 8 shows a partial example of a state variable description. Again, our desire to enforce completeness requires that state variable definitions include such information as arrival rates, exceptional condition handling, data age requirements, feedback information, etc. not shown here.

SpecTRM-RL requires all state variables that describe the process state to include an *unknown* value. This value is the default value upon startup or upon specific mode transitions (for example, after temporary shutdown of the computer). This feature is used to ensure consistency between the computer model of the process state and the real process state by forcing resynchronization of the model with the outside world after an interruption of processing. Many accidents have been caused by the assumption that the process state does not change

State

Supervisory-Interface
↳ Pilot-Displays
↳ Resolution-Advisory
↳ **Vertical-Control**

DEFINITION

= Blank

INITIALLY

= Other

Some RA-Strength$_{\text{s-277}}$ **in state** Increase-2500fpm	F	F	F
Some Reversal$_{\text{s-282}}$ **in state** Reversed	F	F	F
Composite-RA$_{\text{s-266}}$ **in state** Climb	F	•	•
Composite-RA$_{\text{s-266}}$ **in state** Descend	F	•	•
Corrective-Climb$_{\text{s-263}}$ **in state** Yes	•	F	•
Corrective-Descend$_{\text{s-264}}$ **in state** Yes	•	•	F
Crossing-Geometry$_{\text{m-388}}$	F	F	F

= Increase

Some RA-Strength$_{\text{s-277}}$ **in state** Increase-2500fpm	T
Climb-Inhibit$_{\text{s-243}}$ **in mode** Inhibited	T
Descend-Inhibit$_{\text{s-245}}$ **in mode** Inhibited	T

= Crossing

Some Reversal$_{\text{s-282}}$ **in state** Reversed	F	F	F	F
Composite-RA$_{\text{s-266}}$ **in state** Climb	T	T	•	•
Composite-RA$_{\text{s-266}}$ **in state** Descend	•	•	T	T
Some RA-Strength$_{\text{s-277}}$ **in state** Increase-2500fpm	F	F	F	F
Corrective-Climb$_{\text{s-263}}$ **in state** Yes	F	•	F	•
Corrective-Descend$_{\text{s-264}}$ **in state** Yes	•	F	•	F
Crossing-Geometry$_{\text{m-388}}$	F	F	F	F

= Maintain

Composite-RA$_{\text{s-266}}$ **in state** Climb	T	•
Some RA-Strength$_{\text{s-277}}$ **in state** Increase-2500fpm	F	F
Composite-RA$_{\text{s-266}}$ **in state** Descend	•	T
Corrective-Climb$_{\text{s-263}}$ **in state** Yes	F	F
Corrective-Descend$_{\text{s-264}}$ **in state** Yes	F	F

= Reversal

Some Reversal$_{\text{s-282}}$ **in state** Reversed	T	T	T
Composite-RA$_{\text{s-266}}$ **in state** Climb	F	•	•
Composite-RA$_{\text{s-266}}$ **in state** Descend	F	•	•
Corrective-Climb$_{\text{s-263}}$ **in state** Yes	T	•	T
Corrective-Descend$_{\text{s-264}}$ **in state** Yes	•	•	T

Fig. 8. Example of SpecTRM-RL state variable specification

while the computer is not processing inputs or by incorrect assumptions about the initial value of state variables.

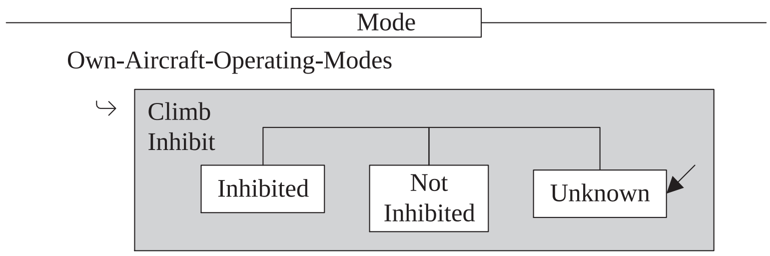
Unknown is used for state variables in the supervisory interface model only if the state of the display can change independently of the software. Otherwise, such variables must specify an initial value (e.g., blank, zero, etc.) that should be sent when the computer is restarted.

In the example shown, *vertical control* is a state variable in the supervisory interface model and is one of the pieces of information displayed to the pilot as part of an RA (Resolution Advisory, which is the escape maneuver the pilot is to implement to avoid the intruder aircraft). Vertical control can have the values *Unknown, Other, Increase, Crossing, Maintain*, or *Reversal.* AND/OR tables are used to specify which of these values is displayed to the pilot (given the current state of the aircraft model and the intruder aircraft being avoided). For example, *Maintain* is displayed if the Composite-RA state variable is in state "Climb", there is no RA-Strength in state "Increase-2500fpm", and Corrective-Climb and Corrective-Descend are both not in state "yes" *or* Maintain is displayed if there is no RA-Strength in state Increase-2500fpm, Composite RA is in state Descend, and again both Corrective-Climb and Corrective-Descend are not in state "yes." Timing constraints may also be specified as conditions in the tables (i.e., conditions on the state transitions) but are not required in this example.

State Transition Specification. As with all state-machine models, transitions in the three parts of a SpecTRM-RL model are governed by external events and the current state of the modeled system. In SpecTRM-RL, the conditions under which transitions are taken are specified separately from the graphical depiction of the state machine. We have found that the behavior of real systems is too complex to write on a line between two boxes. Instead, we again use AND/OR tables. Figure 9 shows an example specification for a transition.

Macros and Functions. Macros are simply named pieces of AND/OR tables that can be referenced from within another table. For example, the macro in Figure 10 is used in the definition of the variable Vertical-Control in Figure 8. The macros, for the most part, correspond to typical abstractions used by application experts in describing the requirements and therefore add to the understandability of the specification. In addition, the abstraction is necessary to handle the complexity of guarding conditions in larger systems and we found this a convenient abstraction to allow hierarchical review and understanding of the specification. Also, rather than including complex mathematical functions directly in the transition tables, functions are specified separately and referenced in the tables. For instance, Own-Tracked-Alt in Figure 9 is a function reference.

The macros and function, as well as the concept of parallel state machines, not only help structure a model for readability; they also help us organize models to enable *specification reuse.* Conditions commonly used in the application domain can be captured in macros and common functions, such as tracking, can be captured in reusable functions. In addition, the parallel state machines allow



DEFINITION

INITIALLY $\longrightarrow$ **Unknown**

true

Unknown, Not-Inhibited $\longrightarrow$**Inhibited**

Composite-RA$_{s\text{-}266}$ **in state** No-RA	T	T
Altitude-Climb-Inhibit$_{v\text{-}259}$ = True	T	•
Own-Tracked-Alt$_{f\text{-}487}$ > Aircraft-Altitude-Limit$_{v\text{-}259}$	T	•
Config-Climb-Inhibit$_{v\text{-}259}$ = True	•	T

Unknown, Inhibited $\longrightarrow$**Not-Inhibited**

Composite-RA$_{s\text{-}266}$ **in state** No-RA	T	T
Altitude-Climb-Inhibit$_{v\text{-}259}$ = True	F	•
Own-Tracked-Alt$_{f\text{-}487}$ > Aircraft-Altitude-Limit$_{v\text{-}259}$	•	F
Config-Climb-Inhibit$_{v\text{-}259}$ = True	F	F

Fig. 9. Example of SpecTRM-RL mode or state transition specification

the internal model of each system component (discussed in Section 3) and the different system modes to be captured as separate and parallel state machines. This helps to accommodate reuse of internal models and operational modes, and helps us plan for product families (research goal 4 in the introduction). Naturally, to accomplish reuse, care has to be taken when creating the original model to determine what parts are likely to change and to modularize these parts so that substitutions can be easily made. This structuring, however, is beyond the scope of the current paper.

Macro

Crossing-Geometry

DEFINITION

Some Crossing$_{\text{s-280}}$ **in state** Int-Cross	T	•
Some Crossing$_{\text{s-280}}$ **in state** Own-Cross	•	T

Fig. 10. Example of SpecTRM-RL macro specification

4 Eliminating Internal Broadcast Events

A third goal for SpecTRM-RL was to eliminate error-prone constructs. During the independent verification and validation (IV&V) of TCAS II, problems with internal broadcast events (used to synchronize parallel state machines in Statecharts and RSML) accounted for a clear majority of the errors related to the syntax and semantics of RSML. Common and difficult to resolve problems involved proper synchronization of mutually interdependent state machines. In addition, getting the state machines to correctly model system startup behavior proved to be surprisingly difficult. Internally generated events seem to cause “accidental complexity” in the specification that is not necessarily present in the problem being specified.

These problems were not just the most common language-related problems in the initial specification, they were also the problems that lingered unresolved (or incompletely resolved) through several cycles of corrections and repeated IV&V. Note that the problems related to synchronization were not directly caused by misunderstandings of the RSML event/action semantics; the event/action mechanism is quite simple and purposely selected to be intuitive [9]. Instead, the problems were caused by the complexity of the model and the inherent difficulty of comprehending the causal relationships between parallel state machines. Thus, this difficulty is not unique to RSML, it is fundamentally difficult to understand parallelism and synchronization. Other state-based languages such as Statecharts

[1] and the UML behavioral (state machine) models that use event/action semantics are likely to encounter the same problem when used to model complex systems. When we eliminated internal events, we were surprised at how much easier it was to rewrite our old specifications (such as TCAS II) and to create new ones.

The trigger events and actions on the transitions in Statecharts and RSML are used for two purposes. First, they are used to sequence and synchronize state machines so the next state relation is computed correctly. For instance, to determine if an intruding aircraft is a collision threat in TCAS II, we must first determine how close the intruder is and how close the intruder is allowed to come before it is considered a threat. Thus, the state variables determining the intruder status and the sensitivity level of TCAS II must be evaluated before we determine advisory-status. This is a straightforward (but as mentioned above, error prone) use of events and actions.

Second, events and actions may be employed to use, in essence, the state machines as a programming language. The events can be used to create loops and counters, and events can be implicitly assigned semantic meaning and used for purposes other than synchronization. In our experience we have found this freedom of using the events a trap that invites the introduction of design details in the specification. During the development of the TCAS II model we had to repeatedly remind ourselves to use events prudently; we have found that even experienced users of such modeling languages inevitably fall into the trap of using events and actions to introduce too much design in the models.

To solve this problem in SpecTRM-RL, we simply decided not to use internal events and instead to rely on the data dependencies in the model to determine the order in which transitions are taken, i.e., the ordering, if critical, is explicitly included in the model as opposed to being built into the semantics of the modeling language. In this way, the reviewer need not rely on knowledge about the semantics of the modeling language to determine if the model correctly matches the intended functional behavior—that behavior (which state transitions follow which) is explicitly specified in the constructed model. A similar argument holds for the modeler. We found that different reviewers of our TCAS specification were assigning differing semantics to the state transition ordering. In the example above, the transitions in the state machine advisory status refer to the states of intruder status and sensitivity level; thus, intruder status and sensitivity level will be evaluated before advisory status. This sequencing will assure a correct evaluation of the next state relation based on the data dependencies of the transitions and variables. The next state function is recomputed every time the environment changes an input variable. Naturally, a SpecTRM-RL specification cannot include any cycles in the data dependencies. Cycles in a specification can be easily detected by our tools.

In Statecharts and RSML, a transition is not taken until an explicit event is generated. When the transition is taken, additional events may be generated as actions. In this way, the events propagate through the state machine triggering transitions. In our formalization of the semantics of RSML [2] we view each transition as a simple function mapping one state to the next. The events and actions on the transitions are used to determine in which order we shall use these functions to compute the next state. We define the new semantics of SpecTRM-RL in essentially the same way as for RSML. The only difference is how we determine in which order to apply the functions representing transitions. We now rely entirely on the data dependencies between the transitions to determine a partial order that is used during the computation of the next state relation. The semantics of SpecTRM-RL have been defined formally, but this definition is not included for space reasons.

5 Conclusions

In this paper, we described some lessons learned during experimentation with a formal specification language (RSML) and how we have used what we learned to drive further research. We showed how a formal modeling language can be designed to assist system understanding and the requirements modeling effort. We achieve this by grounding the design of the language in the domain for which it is intended (process control) and how people actually think about and conceptualize complex systems.

We have applied these principles to the design of a new experimental language called SpecTRM-RL. As mentioned above, SpecTRM-RL evolved from our previous experiences with using RSML to specify large and complex systems. In particular, we addressed the problems associated with inclusion of excessive design in the blackbox specification and internal broadcast events. Our experience thus far indicates that the new language design principles introduced in SpecTRM-RL greatly enhance the usability of a formal notation.

References

1. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.
2. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, pages 363–377, June 1996.
3. C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions of Software Engineering and Methodology*, 5(3):231–261, July 1996.
4. K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Januaray 1980.
5. Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human–Computer Interaction*, 1:311–338, 1985.
6. Matthew S. Jaffe, Nancy G. Leveson, Mats P.E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

7. D.J. Keenan and M.P.E. Heimdahl. Code generation from hierarchical state machines. In *Proceedings of the International Symposium on Requirements Engineering*, 1997.
8. N.G. Leveson. *Safeware: System Safety and Computers.* Addison Wesley, 1995.
9. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, pages 684–706, September 1994.
10. N.G. Leveson, J.D. Reese, S. Koga, L.D. Pinnel, and S.D. Sandys. Analyzing requirements specifications for mode confusion errors. In *Proceedings of the Workshop on Human Error and System Development*, 1997.
11. E.I. Lowe. *Computer Control in Process Industries.* Peregrinus, 1971.
12. Robyn R. Lutz. Targeting safety related errors during software requirements analysis. *Journal of Systems Software*, 34(3):223–230, September 1996.
13. David L. Parnas. Tabular representations of relations. Technical Report CLR report No. 260, McMaster University, Hamilton, Ontario, October 1992.
14. David L. Parnas and Jan Madey. Functional documentation for computer systems engineering (volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.