Completeness in Formal Specification Language Design for Process-Control Systems*

Nancy Leveson
Massachusetts Institute of Technology
Aeronautics and Astronautics Dept.
77 Massachusetts Ave.
Cambridge, MA 02139
617-258-0505
leveson@mit.edu

Abstract: This paper examines the issue of completeness in specification language design. In the mid-80s we identified a set of 26 formal criteria to identify missing, incorrect, and ambiguous requirements for process-control systems. Experimental validation of the criteria on NASA and NASDA spacecraft systems have supported their usefulness in detecting commonly omitted but important information and engineers have been using them in checklist form on real systems. At the same time, we have extended the criteria and now have over 60. This paper shows how most of the criteria can be embedded in a formal specification language in ways that potentially allow automated checking or assist in manual reviews.

Keywords: Formal specification language, completeness, process-control

1 Introduction

As part of a research effort to investigate the general problem of providing tools to assist in developing embedded systems, we have been experimenting with formal specification language design. Our latest experimental toolset is called SpecTRM (Specification Tools and Requirements Methdology), and the formal specification language is SpecTRM-RL (SpecTRM-Requirements Language).

The design of SpecTRM-RL is based upon lessons learned from experimentation with previous language design features. Our goal is to build knowledge incrementally about how to most effectively design specification languages by applying the following research paradigm: (1) determine important goals for specification languages from experience with industrial applications, (2) generate hypotheses about how these goals might be accomplished, and (3) instantiate these hypotheses in the design of a specification language that we use in further experimentation to generate more goals and hypotheses and better languages.

In our first attempts at learning about formal specification language design, 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, TCAS II, used on commercial passenger aircraft [9]. We have taken what we learned from our use of RSML, particularly the limitations of the language, and generated goals for SpecTRM-RL and hypotheses about potential design features that will support these goals.

The new goals and hypotheses concern (1) enhancing readability and reviewability, (2) eliminating what we found to be error prone features, such as internal broadcast events, (3) writing pure blackbox specifications, (4) designing to allow reuse and specification of program families, and (5) eliminating common specification flaws such as incompleteness. In a previous paper, we described what we learned from our experiences with RSML about the first four goals and how we applied these lessons to the design of SpecTRM-RL [10]. This paper describes the lessons learned and our attempts to respond to them with respect to the fifth goal—providing languages that allow and encourage completeness in specifications.

The next two sections describe the rationale behind the goal and what has been done previously. The rest of the paper describes how we have approached the problem of specification completeness in the design of SpecTRM-RL.

2 The Completeness Problem

Accidents and major losses involving computers are usually the result of incompleteness or other flaws in the software

*To appear in Formal Methods in Software Practice, August 2000, Portland. This research was partially supported by a grant from NASA Langley Research Center.

requirements, not coding errors [7, 12]. But even for non-critical systems, incompleteness appears to be an important aspect of software correctness. Cognitive psychologists have determined that people tend to ignore information during problem solving that is not represented in the specification of the problem. In experiments where some problem solvers were given incomplete representations while others were not given any representation at all, those with no representation did better [1, 13]. An incomplete problem representation actually *impaired* performance because the subjects tended to rely on it as a comprehensive and truthful representation—they failed to consider important factors deliberately omitted from the representations [14].

One possible explanation for these results is that some problem solvers did worse because they were unaware of important omitted information. However, both novices and experts failed to use information left out of the diagrams with which they were presented, even though the experts could be expected to be aware of this information. Fischoff, who did such an experiment involving fault tree diagrams, attributed it to an "out of sight, out of mind" phenomenon [1].

An application for which completeness is particularly critical is process-control, i.e., software that controls arbitrarily large or energetic physical phenomena. Such software is usually real-time and often embedded within some larger system such as a ship, aircraft, missile, spacecraft, manufacturing or processing plant, or transportation system. Its role within the embedded system is to assist in the formulation and implementation of automated or human decisions in controlling the larger system. In such process-control systems, minor behavioral distinctions often have significant consequences. It is therefore particularly important that the requirements specification distinguish the behavior of the desired software from that of any other, undesired program that might be designed; that is, it must be precise (unambiguous), correct with respect to the encompassing system requirements, and complete.

Absolute completeness, of course, is a theoretical but probably unattainable goal for most specifications—the only truly complete specification of something would be the thing itself. It also may be unnecessary and uneconomical for most situations. The trick is to determine whether a specification is *sufficiently* complete—it differentiates between desired and undesired implementations with respect to properties that the system designers (and the customers) care about.

Sufficient needs to be determined, then, with respect to the overarching system requirements. There is widespread agreement that requirements specifications should be blackbox (*what* vs. *how*), i.e., they should specify only externally observable behavior and qualities, leaving the implementors with as much freedom as possible in making implementation design decisions. But what blackbox behavior should be included is a function of the overall operational system requirements—the blackbox behavior has to matter from an engineering viewpoint for it to be worthwhile specifying. For safety-critical systems, sufficient completeness may be defined in terms of system safety design constraints as well as requirements, which may in turn be derived from a hazard analysis.

Determining whether the operational system requirements and safety-related design constraints are complete or correct is a system engineering problem. However, if a formal specification language does not allow specifying all the important behavioral characteristics identified in or implied by the system specification, then the specification will, by definition, be incomplete. In addition, because traceability is the primary way that engineers ensure that subsystem requirements specifications and implementations are complete with respect to the specified system requirements and design constraints, our specification methodologies must support traceability.

3 Completeness Criteria and the RSM

It is often suggested that simply using a formal specification language will lead to finding flaws, particularly incompleteness in the specifications. This hypothesis has never been validated experimentally. From an intuitive standpoint, its truth will depend on the features of the formal specification language. The author has seen many formal requirements specifications that omit extremely important information for the developers and reviewers. Some languages do not even *allow* specifying important types of software behavioral requirements.

We started investigating the problem of completeness in requirements in the mid-80's after discovering how important it was and how little about it was documented in the literature. Jaffe wrote a dissertation on the topic [5]. Using this research, we defined a set of 26 formal criteria to identify missing, incorrect, and ambiguous requirements for process-control systems [6]. Engineers have made these criteria into checklists and used them on a wide variety of applications, such as radar systems, the Japanese module of Space Station Freedom, and review criteria for FDA medical device inspectors. The feedback we have received from industry is that the criteria are very useful in finding important requirements flaws, but that they are difficult to apply in a checklist format. 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 accomplished so far with respect to these goals.

Through our research, we have extended the original 26 criteria to nearly 60—many of the recent additions are related to human–computer interaction and mode confusion. Note that completeness in engineering specification includes more than simply completeness with respect to a mathematical theory or formal logic but also includes completeness with respect to human (cognitive engineering), application,

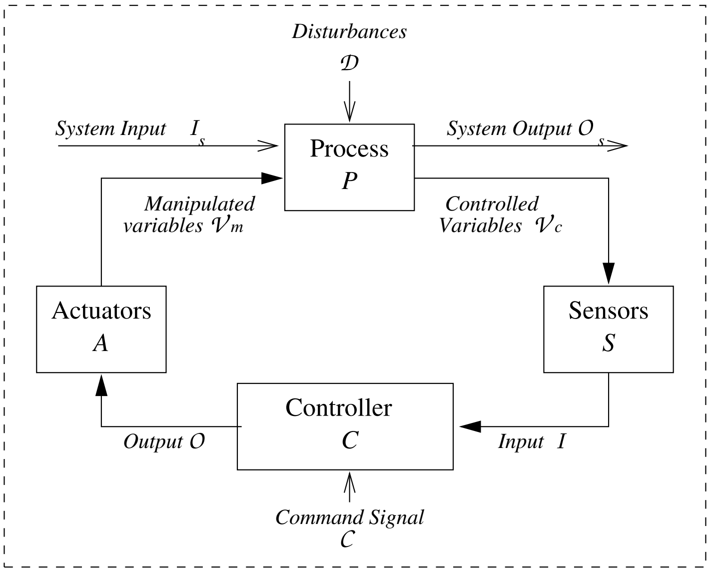

Figure 1: A Classic Control Loop

engineering, and other factors. Space limitations make it impossible to describe all the criteria here—the interested reader is referred to [7, 11]. This section describes the criteria in general and the underlying formal model on which the criteria are defined. The next section describes how the completeness criteria are embedded within the SpecTRM-RL language design.

Our completeness criteria were defined to be applicable to any behavioral requirements specification. We also wanted to relate the criteria to general process-control concepts. To accomplish these goals, we defined a general behavioral model of the control function, called a requirements state machine (RSM) [6]. The RSM is based on a simple Mealy automaton with outputs on the transitions between states, which is also the underlying model for SpecTRM-RL (and many other languages). Transitions are labeled with logical expressions of the form *Input predicate*$||$*Output predicate*, and a transition is taken if the *Input predicate* on that transition evaluates to **true**. If an output is to be produced, the constraints on that output are expressed in the *Output predicate* associated with the transition.

The RSM is very low-level and not appropriate as a modeling language for complex systems—SpecTRM-RL acts as a specification language that overlays the low-level formal model. As long as the mapping from SpecTRM-RL to the RSM is unambiguous and well-defined, formal analysis is possible on both the underlying RSM formal model as well as the higher-level SpecTRM-RL specification itself.

In deriving many of our completeness criteria, we mapped the parts of the RSM automaton to the parts of a process control loop (see Figure 1). Using basic process-control concepts as well as past accidents and incidents, we then defined semantic completeness criteria and heuristics in terms of the parts of the state machine [6] using properties of the process, sensor, and actuator functions. These criteria include absolute criteria that must be satisfied by the software requirements specifications for all such systems in order to be adequately complete for a safety-critical system and also heuristics that will aid in finding missing requirements, incorrect requirements, and ambiguous requirements when considered within the context of the particular goals and constraints of the system being developed.

The RSM is defined as a seven-tuple (Σ, Q, q_0, P_t, P_o, γ, δ) where:

- Σ is the set of input and output variables, $\mathcal{I}$ and $\mathcal{O}$, to the controller software. Completeness criteria here ensure that all sensor input is used by the controller. Additional criteria require that legal output values never produced by the computer are identified and investigated for potential incompleteness.

- Q is the finite set of states of the controller C and q_0 is the initial state of C. Completeness criteria defined on Q and q_0 primarily ensure that startup and shutdown behavior is completely and correctly specified. Most accidents occur during startup or shutdown or during transitions from normal to non-normal operation and between various operating modes, particularly off-nominal modes.

- P_t is the set of Boolean functions over Σ; they represent predicates on the values and timing of the inputs ($\mathcal{I}$) from the sensors. These predicates are called *trigger* predicates because they trigger a state change in the RSM. Criteria here ensure the complete description and handling of inputs including essential value and timing assumptions and robustness in the system implementation. A *robust* system will detect and respond appropriately to violations of assumptions about the system environment (such as unexpected inputs). Robustness with respect to a state-machine description implies the following:

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

  Thus, the software must be prepared to respond in real time to all possible inputs and input sequences—there must be no observable events that leave the program's behavior indeterminate. These criteria are the most closely related to the mathematical completeness often checked by formal methods tools (for example, in SCR [4] and RSML [3]).

  Although these robustness criterion ensure that there is always one transition that can be taken out of every state, they do not guarantee that all assumptions about the environment have been specified or that there is a defined response for all possible input conditions the environment can produce. Many of these assumptions and conditions are application dependent (e.g., the conditions under which reverse thrusters should be fired), but some are essential for all systems; some of our other criteria ensure the inclusion of this essential information. Timing and load assumptions are especially critical here, and they are often left out of or left incomplete in requirements specifications.

- P_o is the set of Boolean functions over Σ; they represent predicates on the outputs ($\mathcal{O}$) of the controller software. Criteria ensure the complete specification of value and timing requirements for outputs including some special requirements for the specification of environmental capacity, data age, and latency.
- γ is the trigger-to-output relationship mapping from $Q \times P_t$ to P_o. That is, $\gamma(q, p)$ gives the predicate describing the output $\mathcal{O}$ to the actuators to be generated when the transition with input predicate p is taken out of state q. Criteria here relate not to input or output predicates alone but to their relationship. These include requirements in most process-control systems for graceful degradation and for using feedback information.
- δ is the state transition function mapping $Q \times P_t$ to Q. That is, $\delta(q, p)$ where $q \in Q$ and $p \in P_t$ defines the next state when the software is in state q and takes the transition having p as the input predicate. Completeness with respect to the state transition function involves properties of the paths between states and the predicate sequences describing these paths. Few absolute criteria can be identified as with the other parts of the RSM; most of the criteria are application-dependent. However, general properties or heuristics can be identified that, together with application-specific information, can be used to guide the analysis process. These include properties associated with basic reachability, recurrent or cyclic behavior, reversible behavior, reachability of safe states, preemption of transactions, path robustness, and general analysis of consistency with required system-level constraints.

In addition the RSM has the following properties:

- Predicates in P_t and P_o are expressed using the standard Boolean operators and ordinary arithmetic operators. The expression $X\uparrow$ represents an input or output occurrence of X. This expression evaluates to **true** the moment input X arrives at the black-box boundary or output X is produced and presented at the black-box boundary. The value of a variable X is denoted $v(X)$.
- When an input I arrives at the software boundary, i.e. an $I\uparrow$ event has occurred, it is denoted as I_j or simply I. The previous occurrence of the same input is denoted I_{j-1} and so forth. The ordering of outputs is expressed in the same manner. Note that the first variable I arriving at the black-box boundary is referred to as I_1.
- A clock and a function giving the absolute time of an event are needed to express timing. The expression $t(I\uparrow)$ denotes the time when I arrives at the black-box boundary. The clock is started when the system receives the signal to startup, i.e. $t(Su\uparrow) = 0$.

4 Intent Specifications, SpecTRM-RL, and Completeness

A great deal more is required in a complete specification than is (or probably can be) included in formal specification languages. Much of this information is most effectively conveyed informally anyway. For example, a person completely unfamiliar with TCAS could not pick up our specification and understand how TCAS works. This general problem of specification completeness in terms of necessary content and structure has been tackled in our research on Intent Specifications, a five-level structure based on intent abstraction. Intent specifications are described elsewhere [8]. In this paper, we include only unpublished results on the formal part (Level 3) of an Intent Specification.

The levels and parts of a complete system Intent Specification are linked together with pointers. As stated earlier, to be a practical requirements specification language, the language must include traceability. Intent specifications include both forward and backward traceability from high-level requirements to code and vice versa. In addition, unlike most specification languages, intent specifications include traceability to and from the hazard analysis so that reviewers can verify that safety-critical constraints are enforced in the software and changes in the code can be identified and analyzed as to their potential effect on safety.

The design of SpecTRM-RL is greatly influenced by our desire to provide a combined specification and modeling language. System specifications (particularly requirements specifications) need to be reviewed and used by people with a large variety of backgrounds and expertise, most of whom are not computer scientists or trained in formal logic or discrete math and may not even be engineers. Therefore, it is not practical to use most formal modeling languages as specification languages. In addition, industrial projects rarely have the resources to provide a separate modeling effort for the specification, and the continual changes common to most software development projects will require frequent updates to ensure that the formal model is consistent with the current requirements and system design.

SpecTRM-RL was designed to satisfy both objectives: to be easily readable enough to serve as part of the official specification of the blackbox behavioral requirements and, at the same time, to have an underlying formal model that can be executed and subjected to mathematical analysis. To assist in readability, we use graphical, tabular, symbolic, and structural notations where we have found each most appropriate for the type of information being specified. Decisions about how things are specified were based on the research literature on visualization, feedback from users of RSML and SpecTRM-RL, our own attempts to build specifications for real systems using the language, and observation of the notations engineers use for specifying these properties.

The SpecTRM-RL notation is driven by the intended use of the language to define a blackbox 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 necessary for specifying component design but not necessary to specify the input/output function computed in a pure blackbox requirements specification. Some features are also the result of wanting to remain as close as possible to the way engineers draw and define control loops.

Figure 2 shows the four main components of a SpecTRM-RL specification: (1) a specification of the supervisory modes of the controller being modeled, (2) a specification of its control modes (3) a model of the controlled process (or *plant* in control theory terminology) that includes the inferred operating modes and system state (these are inferred from the measured inputs), and (4) a specification of the inputs and outputs to the controller. The graphical notation mimics the typical engineering drawing of a control loop.

Every automated controller has at least two interfaces: one with the supervisor(s) that issues instructions to the automated controller (the supervisory interface) and one with each controlled system component (controlled system interface). The supervisory interface is shown to the left of the main controller model while the interface with the controlled component is to shown the right.

The supervisory interface consists of a model of the operator controls and a model of the displays or other means of communication by which the component relays information to the supervisor. Note that the interface models are simply the logical view that the controller 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. By separating the assumed interface from the real interface, we are able to model and analyze the effects of various types of errors and 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. During development, mockups of the physical screen or interface design can be generated and tested using the output of the SpecTRM-RL simulator.

The bottom left quadrant of Figure 2 provides information about the operating modes for the controller itself. These are not internal states of the controller (which are not included in our specifications) but simply represent externally visible behavior about the controller's modes of operation (described further below).

The right half of the controller model represents inferred information about the operating modes and states of the controlled system (the *plant* in control theory terminology). A simple plant model may include only a few relevant state variables. If the controlled process or component is complex, the model of the controlled process may 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 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 can be inserted into the SpecTRM-RL process model. Parts of a SpecTRM-RL model can be reused or changed to represent different members of a product family.

Figure 3 shows the graphical part of a SpecTRM-RL specification of a simple altitude switch. The specification is based on an unpublished specification of an altitude switch by Steve Miller at Rockwell Collins. This switch turns on a Device of Interest (DOI) when the aircraft descends through a threshold altitude.

In SpecTRM-RL, state values in square boxes in the right side of the diagram represent inferred values used in the control of the computation of the blackbox I/O function. Such variables are necessarily discrete in value[1], and thus can be represented as a state variable with a finite number of possible values. In practice, such state variables almost always have only a few relevant values (e.g., altitude below a threshold, altitude at or above a threshold, cannot-be-determined, and unknown). Values for state variables in the plant model are required in SpecTRM-RL to include an *unknown* value. The meaning and purpose of the unknown state value are described below.

In the altitude switch example, defining the control algorithm requires using information about the aircraft altitude level with respect to a given threshold, the inferred status of the DOI, and the validity of the altimeter information being provided as well as the measured variables and various constants defined elsewhere in the specification.

The possible values for a state variable are shown with a line connecting the boxes. The line simply denotes that the values are disjoint, that is, the variable may assume only one value at a time. A small arrow pointing at a box denotes the default (startup) value for the state variable or mode. For example, the DOI-Status can have the values *On, Off, Unknown*, and *Fault-Detected*. The default value is *Unknown*.

The altitude switch has two control inputs (shown on arrows to the left of the Component diagram): a reset signal that has the value *true* or *false* and an inhibit button that inhibits operation of the altitude switch. The inhibit button

[1] 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.

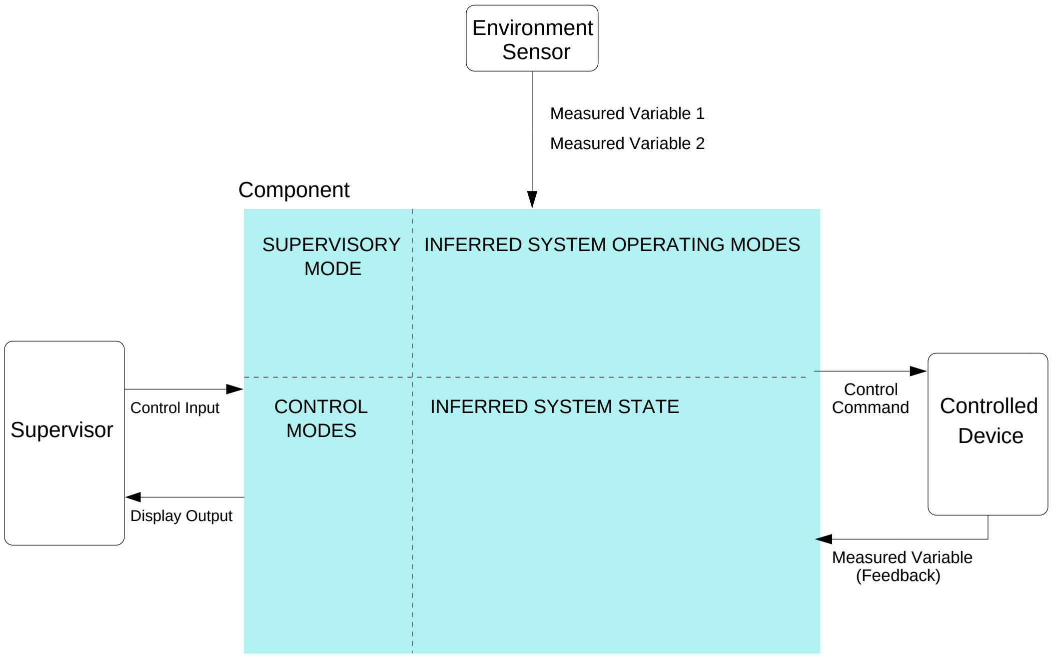

Figure 2: The Parts of a SpecTRM-RL Specification.

can either be in the *on* or *off* position. The only display in the altitude switch example is a fault indicator lamp that can also either be *on* or *off*, but its content is controlled through the watchdog timer and not directly by the altitude switch. There is only one supervisory mode—cockpit controlled—which is shown in the upper left quadrant of the component model.

Inputs representing the state of the plant (monitored or measured variables) are shown with arrows pointing to the controller. For the altitude switch, these variables provide the current status (on or off) of the device of interest (DOI) that the altitude switch turns on and inputs about the status and value of three altimeters on the aircraft (one analog and two digital) that provide information to the altitude switch about the current measured altitude of the aircraft as well as the status of that information (i.e., normal operation, test data, no computed data provided, or failed),

The output commands are denoted by outward pointing arrows. In the example, they include a signal to *power-on* the device (DOI) and a *strobe* to a watchdog timer so that proper action can be taken (by another system component) if the altitude switch fails. The outputs in this example are simple "high" signals on a wire or line to the device.

Note that the internal design of the altitude switch is not included in the model. The altitude switch operating modes are externally visible (and must be known for the pilot to understand its operation) and the aircraft model is used to describe the externally visible behavior of the altitude switch in terms of the process being controlled (and not in terms of its own internal data structures and algorithms). Thus the specification is blackbox.

Because of the simplicity of the altitude switch example, there are a few features of SpecTRM-RL that are not needed or are not well illustrated. Almost all of the missing features involve the ability to specify modes. Modes are abstractions on states and are not necessary for defining blackbox behavior. They are useful, however, in understanding or explaining the behavior of complex systems. While some formal specification languages use the term "mode" as a synonym for state (all modes are states and vice versa), SpecTRM-RL uses the more limited definition of mode common in engineering, i.e., as a state variable that plays a particular role in the state machine. In this usage, modes partition the state space into disjoints sets of states. For example, the state machine may be in normal operational mode or in a maintenance mode. Our definition was chosen to assist in reviewability of the specification by domain experts and in formal analysis of specifications for particular properties commonly involved in operator mode confusion [11]. SpecTRM-RL allows specifying several types of modes: supervisory modes, control modes, controlled-system operating modes, and display modes.

Supervisory modes are useful when a component may have multiple supervisors at any time. For example, a flight control computer in an aircraft may get inputs from the flight management computer and also directly from the pilot. Required behavior may differ depending on which supervisory mode is currently in effect. Mode-awareness errors related to confusion in coordination between multiple supervisors can be defined (and the potential for such errors theoretically identified from the models) in terms of these supervisory modes.

Control Modes control the behavior of the controller itself. Modern avionics systems may have dozens of modes.

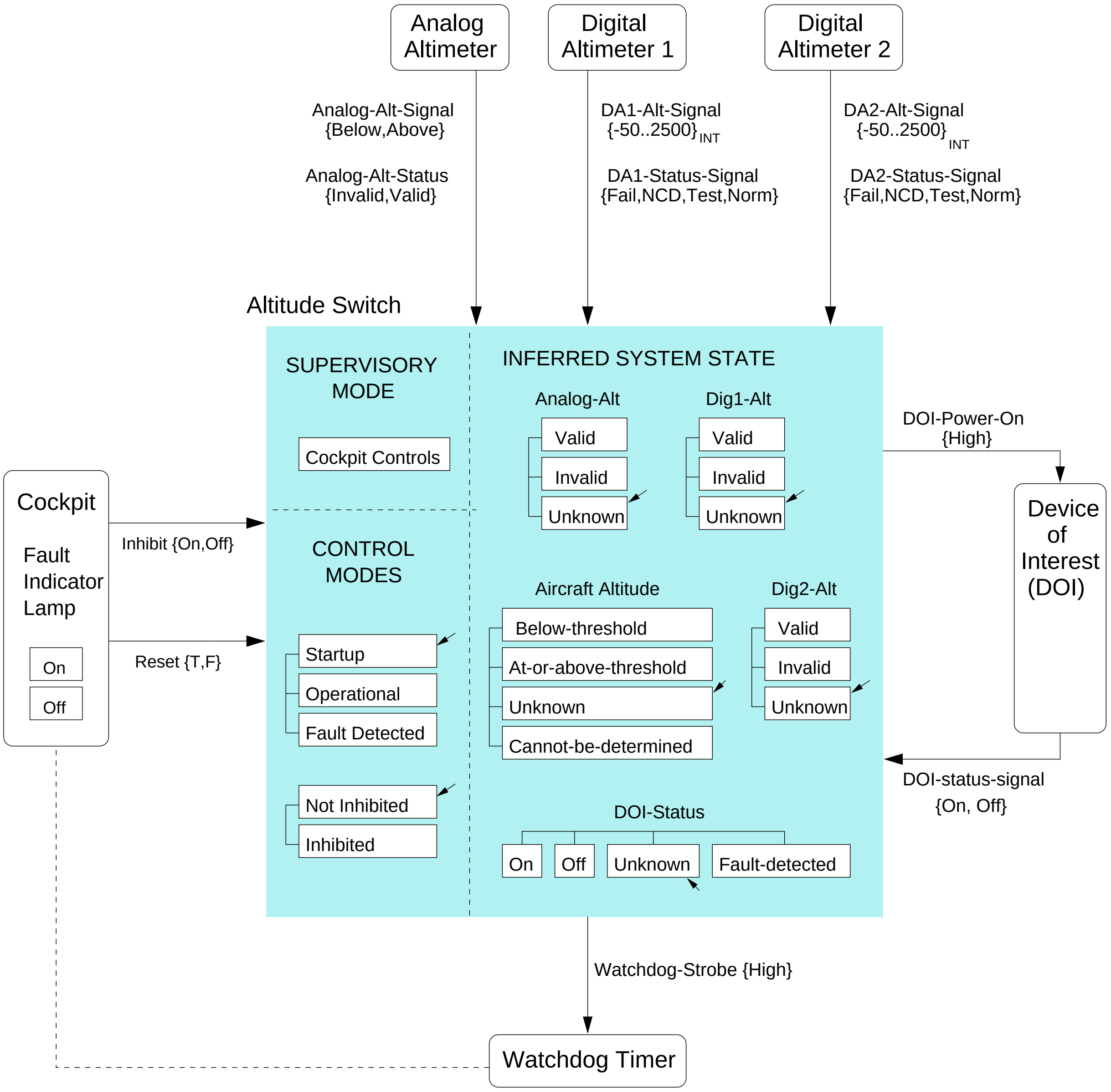

Figure 3: Example of the Graphical Model

Control modes may be used in the interpretation of the component's interfaces or to describe the component's required process-control behavior. In the altitude switch, two types of control modes are useful in specifying the blackbox behavior: (1) whether the switch is in the *startup*, *operational*, and *internal-fault-detected* mode (the latter will result in the fault indicator light being lit in the cockpit and cessation of activity until the reset button is pushed) and (2) whether the operation of the altitude switch is partially inhibited or not. These two sets of modes cannot be combined in this case as they are not disjoint. In fact, in my original specification of the altitude switch, I did combine them and then found a flaw in that specification through the use of our completeness criteria and discovered they needed to be separated.

A third type of mode, *controlled-system* or plant operating modes, can be used to specify sets of related behaviors of the controlled-system (plant) model. They are used to indicate its operational status. For example, it may be helpful to define the operational state of an aircraft in terms of it being in takeoff, climb, cruise, descent, or landing mode. Such operating modes are not needed to define the behavior of the altitude switch and thus are not included in the example.

In systems with complex displays (such as Air Traffic Control systems), it may also be useful to define various *display modes*.

The SpecTRM-RL language is composed of the graphical notation along with specifications for output messages or commands, modes and inferred state variables, input and output variables, macros, and functions. The graphical notation has been described. Each of the other features is now described with emphasis on how their specification relates to our completeness criteria.

4.1 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. This relationship is the most critical in understanding and reviewing a system requirements specification, and therefore saliency of this information can assist in these tasks. Other state-machine specification languages, such as RSML and Statecharts, do not explicitly show this relationship, although it can be determined, with some effort, by examining the specification.

The general form of an output specification is shown in Figure 4. The following information can and should be included: **destination** of the output; **acceptable values**; **timing behavior** including any initiation delay or completion deadline along with any required exception-handling behavior if the deadlines cannot be met, output load and capacity limitations, etc.; **feedback information** about how the controller will determine that the output command has been successfully implemented (see "Output to Trigger Event Relationships" in Chapter 15 of *Safeware*); and the identity of any other output commands that **reverse** this output.

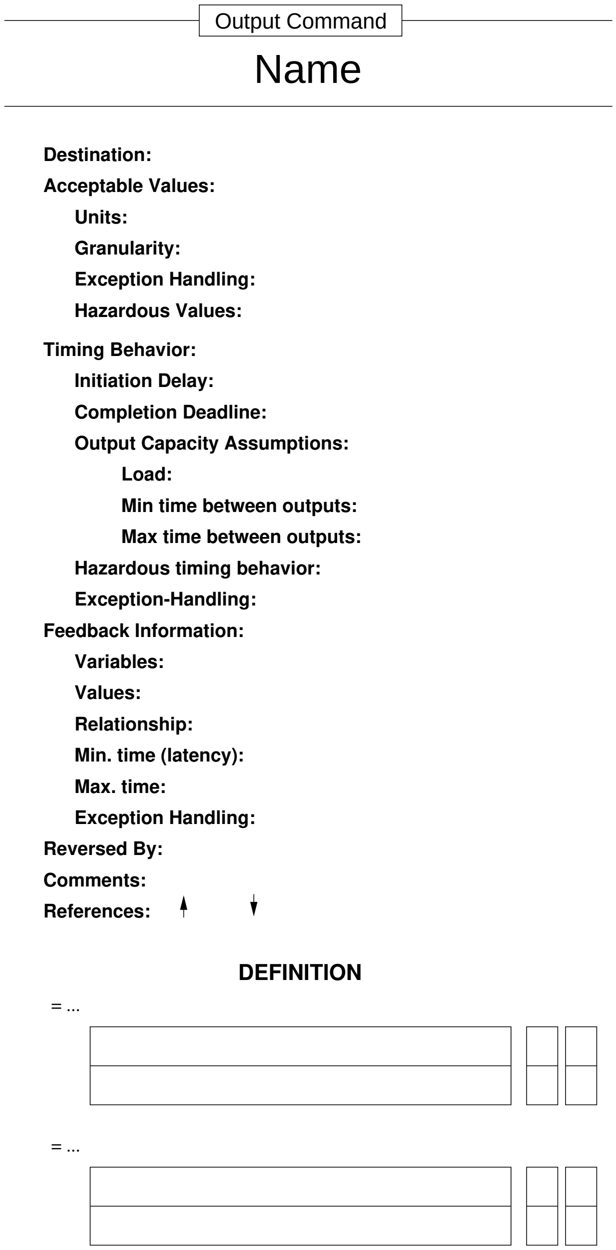

Figure 4: The General Format of an Output Specification

In many systems, it is important to indicate a maximum time in which the output command remains effective before it is executed. After this time, the output essentially "times out" and should not be executed. This *data age* information can either be provided in the output message (if the timeout is implemented by the output command actuator) or included in the reversal information if the controller must issue a reversal to undo the command's effect. Reversal information is also useful in identifying accidentally omitted behavior from the specification, i.e., most output actions to provide a change in the controlled plant or supervisory interface have complementary actions to undo that change. **References** are pointers to other levels of the intent specification related to this part of the specification and are used for traceability.

Some of this information may not be applicable or important for a particular system. However, by including a place for it in the specification language syntax, the specifier must either include it or indicate that the information is not applicable or important. The implementors and maintainers need to know that these behaviors are not important and why not and also need to know that it was considered by the original specifiers and not simply forgotten. Lots of accidents and incidents result from such a lack of consideration of these factors by designers and implementors.

The conditions under which an output is triggered (sent) can be specified by a predicate logic statement over the various states, variables, and modes in the specification. In our experience in specifying complex systems, however, we found that the triggering conditions required to accurately capture the requirements are often extremely complex. We also found propositional logic notation did not scale well to complex expressions in terms of readability and error-proneness. To overcome this problem, in RSML we developed a tabular representation of disjunctive normal form (DNF) that we call AND/OR tables[2].

The far-left column of the AND/OR table lists the logical phrases of the predicate. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns evalutes to *true*, then the entire table evaluates to *true*. A column evaluates to *true* if all of its elements match the truth values of the associated predicates. A dot or empty square denotes "don't care."

For SpecTRM-RL, we kept the very successful AND/OR tables but made some changes. The tables are now divided into two parts: an upper part denoting the relevant control modes for that column and a lower part describing any additional conditions for triggering the output. We have found that this separation assists in completeness checking, particularly when humans are writing and reviewing specifications. For completeness reasons, every output command column must include a reference to the control mode(s) under which the command is sent. It is assumed that if not specified, then the output cannot occur in that mode.

For the altitude switch *DOI-Power-On* output in the example shown in Figure 5, the command is triggered (sent) when all the following conditions are true: the altitude switch is in the operational and not-inhibited modes, the DOI is not on, the altitude is below the threshold, and the previous altitude was at or above the threshold (the requirements for the altitude switch say that if the switch is turned off while the aircraft is below the threshold altitude, the DOI is not powered on again until the aircraft goes above the threshold altitude and again passes down through it). The *Prev* built-in function—a feature of the underlying formal RSM model—allows referring to previous values of modes, state variables, inputs, and outputs. References to time are also allowed in the specification of trigger conditions. An example is shown later.

The specification of constant values represents a tradeoff design decision in most specification languages. There are two choices: (1) use the constant value everywhere it is referenced, and when changes are made all these locations must be identified and updated or (2) assign an identifier name to the constant, use that identifier (instead of the constant) throughout the specification, and have the reader look up the current value in a table somewhere.

If the constant values, such as the altitude threshold, are included directly in the specification (rather than included in a table somewhere other than where it is used), the specification will be easier to read and review because the reviewer will not need to continually flip to another page where the constant is defined when reading the specification. On the other hand, changing the value of the constant when the requirements change can then be tricky and error prone. Using a table simplifies the updating problem because the change need only be made in one place.

We provide a compromise solution to this specification language design problem that satisfies both the readability and changability requirements. The constant itself is used throughout the specification with a subscript that attaches an identifier to it (for an example, see Figure 8). The constant identifiers are maintained in a table in the specification, but the reviewer need not refer to this table (it is only included for convenience). When a constant is changed, for example, the altitude threshold is changed from 2000 to 2500, the tools can automatically search for instances in the specification and update them all.

4.2 Input Variable Definition

Our desire to enforce completeness in the language itself (to satisfy our completeness requirements) leads to language features that allow the inclusion of information (if relevant) about input arrival rates, exceptional-condition handling, data-age requirements, etc. No input data is good forever; after some point in time it becomes obsolete and should not be

[2] For those familiar with other state-machine specification languages that use tables, such as SCR, it is useful to note that tables are used very differently here. In such languages, the actual transitions between states are described in a table. We do not do this in SpecTRM-RL; instead, the tables are used simply to represent one predicate logic statement about the conditions on one transition arrow between states. Therefore our tables are of much more limited size and we have found that their use scales up to very large and complex system specifications while still remaining relatively small.

Output Command

DOI-Power-On

Destination: DOI

Acceptable Values: {high}

Initiation Delay: 0 milliseconds

Completion Deadline: 50 milliseconds

Exception-Handling: (What to do if cannot issue command within deadline time)

Feedback Information:

Variables: DOI-status-signal

Values: high (on)

Relationship: Should be on if ASW sent signal to turn on

Min. time (latency): 2 seconds

Max. time: 4 seconds

Exception Handling: DOI-Status changed to Fault-Detected

Reversed By: Turned off by some other component or components. Do not know which ones.

Comments: I am assuming that if we do not know if the DOI is on, it is better to turn it on again, i.e., that the reason for the restriction is simple hysteresis and not possible damage to the device.

This product in the family will turn on the DOE only when the aircraft descends below the threshold altitude. Only this page needs to change for a product in the family that is triggered by rising above the threshold.

References: ↑ 2.33 ↓ 4.84

CONTENTS

= discrete signal on line PWR set to high

TRIGGERING CONDITION

Control Mode	Operational	T
	Not Inhibited	T
State Values	DOI-Status = On	F
	Altitude = Below-threshhold	T
	Prev(Altitude) = At-or-above-threshold	T

Figure 5: Example of an Output Specification

Input Value

DA1-Alt-Signal

Source: Digital Altimeter 1

Type: integer

Possible Values (Expected Range): -20..2500

Exception-Handling: Values below -20 are treated as -20 and values above 2500 as 2500

Units: feet AGL

Granularity: 1 foot

Arrival Rate (Load): one per second average

Min-Time-Between-Inputs: 100 milliseconds
Max-Time-Between-Inputs: none

Obsolescence: 2 seconds

Exception-Handling:

Description:

Comments:

References: ↑2.11 ↓4.2

Appears in: Altitude

DEFINITION

= FIELD (Altitude in DA1-Message)

Receive DA1-Message FROM Digital-altimeter-1	T

= PREV (DA1-Alt-Signal)

Receive DA1-Message FROM Digital-altimeter-1	F
Time > Time (DA1-Message arrived) + 2 seconds	F

= Obsolete

Receive DA1-Message FROM Digital-Altimeter-1	F	
Time > Time (DA1-Message arrived) + 2 seconds	T	
Powerup		T

Figure 6: Example of an Input Specification

State Value

Altitude

Obsolescence: 2 seconds

Exception-Handling: Because the altitude-status-signals change to obsolete after 2 seconds, altitude will change to Unknown if all input signals are lost for 2 seconds.

Description: When at least one altimeter reports an altitude below the threshold, then the aircraft is assumed to be below the threshold. ↑2.12.1

Comments:

References: ↑2.12 ↓4.10

Appears in: DOI-Power-On

DEFINITION

= Unknown

Startup	T		
Controls.Reset = T		T	
Analog-ALT = Unknown			T
Dig1-Alt = Unknown			T
Dig2-Alt = Unknown			T

The altitude is assumed to be unknown at startup, when the pilot issues a reset command, and when no recent input has come from any altimeter.

= Below-threshold

Analog-Valid-and-Below$_m$	T		
Dig1-Valid-and-Below$_m$		T	
Dig2-Valid-and-Below$_m$			T

At least one altimeter reports a valid altitude below the threshold..

= At-or-above-threshold

Analog-Valid-and-Above$_m$	T	T	T	F	T	F	F
Dig1-Valid-and-Above$_m$	T	T	F	T	F	T	F
Dig2-Valid-and-Above$_m$	T	F	T	T	F	F	T

At least one altimeter reports a valid altitude above the threshold and none below.

= Cannot-be-determined

Analog-Alt = Invalid	T
Dig1-Alt = Invalid	T
Dig2-Alt = Invalid	T

No valid data is received from any altimeter (all report test or failed status).

Figure 7: Example of an Inferred State Variable Specification

used. We provide a special value, *obsolete*, that an input variable assumes a specified time after the last value is received for that variable.

In the example shown in Figure 6, the specification states that the value comes from the *altitude* field in the DA1-message and is assigned when a message arrives. If no message has arrived in the past 2 seconds, the previous value is used. If the last message arrived more than 2 seconds before, the data is considered obsolete. The input variable also starts with the obsolete (undefined) value upon powerup. Because of the similarity of the form of most input definitions, we may simplify the notation in the future.

When the controller has multiple supervisory modes, these must be specified to denote which inputs should be used at any time. The altitude switch example, however, does not have this property.

4.3 State Variable Definition

State variable values are inferred from the values of input variables or from other state variable values. Figure 7 shows a partial example of a state variable description for the altitude switch.

As stated earlier, 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 upon startup or after leaving control modes where processing of inputs has been interrupted. By making *Unknown* the default state value and by assuming the *unknown* value upon changing to a control mode where normal input processing is interrupted (for example, a maintenance mode), the use of an *unknown* state value forces resynchronization of the model with the outside world after an interruption in processing inputs. Many accidents have been caused by the assumption that the process state does not change while the computer is idle or by incorrect assumptions about the initial value of state variables on startup or restart.

If a model of a supervisory display is included in the specification, *unknown* is used for state variables in the supervisory display 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.

4.4 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 8 is used in the definition of the state variable *altitude* in the altitude switch example. Its use is not necessary, but simplifies the specification of altitude and thus makes it easier to understand while also simplifying making

Macro

Dig1-Valid-and-Below

Description:

Comments:

References: ↑2.5.1 ↓4.3, 4.8.2

Appears in: Altitude

DEFINITION

Dig1-Alt = Valid	T
DA1-Alt-Signal < 2000_{THRES}	T

Figure 8: Example of a Macro Specification

changes and enhancing specification reuse. 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, we have found this feature convenient for expressing hierarchical abstraction and enhancing hierarchical review and understanding of the specification. For very complex models (e.g., a flight management system), we have found that macros are almost required for humans to be able to handle the complexity involved in constructing the specification.

Rather than including complex mathematical functions directly in the transition tables, functions may be specified separately and referenced in the tables.

The macros and functions, as well as other features of SpecTRM-RL, not only help structure a model for readability, they also help organize models to enable specification reuse. Conditions commonly used in the application domain can be captured in macros and common functions can be captured in reusable functions. 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.

5 Conclusions and Future Work

This paper has examined the issue of completeness in formal specification language design by showing how such languages can be designed to include necessary information for completeness and correctness and to enforce completeness criteria that are associated with safety and preventing accidents. A few of our completeness criteria, primarily involving heuristic criteria on state sequences, cannot be enforced through language syntax and semantics alone, but most of these extra criteria are easily checked through human review or by the use of relatively simple automated tools.

For example, we define path rebustness in terms of soft and hard failure modes [7]. A *soft failure mode* is one in which the loss of the ability to receive a particular input I *could* inhibit the software from providing an output with a particular value while a *hard failure mode* involves the loss of the ability to receive an input I that *will* prevent the software from producing that output value. One of our completeness criteria related to safety is that:

> *Soft and hard failure modes should be eliminated for all hazard-reducing outputs. Hazard-increasing outputs should have both soft and hard failure modes.*

Soft and hard failure modes can be identified by simply looking at the SpecTRM-RL model itself and do not require searching the entire reachable state space. Properties analyzable directly on the metalanguage for describing the state space (such as SpecTRM-RL) will usually involve simpler tools than those involved in searching reachability graphs. The design of the specification language obviously can affect how easy to use and efficient these analysis tools can be.

SpecTRM-RL is an experimental tool. As such, it is changing as we gain experience with the language and evaluate various hypotheses about how language design can be used to enforce completeness (as well as other research hypotheses). We are currently using it to specify real systems, the largest and most complex being the vertical flight control system for the MD-11 aircraft and the control software for an autonomous helicopter. We are taking what we learn from these empirical evaluations and evolving the language design to generate more hypotheses and language design goals. In this way, we hope to advance the state of knowledge about how to design such languages.

We are also developing new specification language design criteria and analysis tools to expand the types of specification flaws and errors we can detect or prevent. For example, mode confusion is an important new problem in high-tech aircraft and other systems where computers and humans share responsibility for control. Our work on mode-confusion analysis (e.g., [11]) will feed back into the goals for specification language design research.

References

[1] B. Fischoff, P. Slovic, and S. Lichtenstein. Fault trees: Sensitivity of estimated failure probabilities to problem representation. *Journal of Experimental Psychology: Human Perception and Performance*, vol. 4, 1978.

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

[3] Heimdahl, M. P. E. and Leveson, N.G. Completeness and consistency analysis of state-based requirements. *Transactions on Software Engineering*, June 1996.

[4] Heitmeyer, C. “Using the SCR Toolset to Specify Software Requirements.” *Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, Florida, Oct. 1998.

[5] Jaffe, M.S. *Completeness, Robustness, and Safety in Real-Time Software*. Ph.D. Dissertation, University of California Irvine, 1988.

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

[7] Leveson, N.G. Safeware: System Safety and Computers. Addison-Wesley Publishing Co., 1995.

[8] Leveson, N.G.. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. on Software Engineering*, January 2000.

[9] Leveson, N. G., M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, September 1994.

[10] Leveson, N.G., Heimdahl, M.P.E., Reese, J.D. “Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future,” *ACM/Sigsoft Foundations of Software Engineering/European Software Engineering Conference*, Toulouse, September 1999.

[11] Leveson, N.G., Reese, J.D., Koga, S., Pinnel, L.D. and Sandys, S.D. Analyzing requirements specifications for mode confusion errors. In *Proceedings of the Workshop on Human Error and System Development*, 1997.

[12] Lutz, R.R. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 35–46, January 1993.

[13] G.F. Smith. Representational effects on the solving of an unstructured decision problem. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-19, 1989, pp. 1083-1090.

[14] K.J. Vicente and J. Rasmussen. Ecological interface design: Theoretical foundations. *IEEE Trans. on Systems, Man, and Cybernetics*, vol 22, No. 4, July/August 1992.