Primary
IN-03

NCC 2-982

Final Report: Safety Analysis of FMS/CTAS Interactions During Aircraft Arrivals

Nancy G. Leveson

December 27, 1998

This grant funded research on human–computer interaction design and analysis techniques, using future ATC environments as a testbed. The basic approach was to model the nominal behavior of both the automated and human procedures and then to apply safety analysis techniques to these models.

Our previous modeling language, RSML, had been used to specify the system requirements for TCAS II for the FAA. Using the lessons learned from this experience, we designed a new modeling language that (among other things) incorporates features to assist in designing less error-prone human–computer interactions and interfaces and in detecting potential HCI problems, such as mode confusion. The new language, SpecTRM-RL, uses "intent" abstractions, based on Rasmussen's abstraction hierarchy, and includes both informal (English and graphical) specifications and formal, executable models for specifying various aspects of the system. One of the goals for our language was to highlight the system modes and mode changes to assist in identifying the potential for mode confusion.

Three published papers resulted from this research (see attached). The first builds on the work of Degani on mode confusion to identify aspects of the system design that could lead to potential hazards. We defined and modeled modes differently than Degani and also defined design criteria for SpecTRM-RL models. Our design criteria include the Degani criteria but extend them to include more potential problems. In a second paper, Leveson and Palmer showed how the criteria for indirect mode transitions could be

CC202A-3

CASI

applied to a mode confusion problem found in several ASRS reports for the MD-88.

In addition, we defined a visual task modeling language that can be used by system designers to model human–computer interaction. The visual models can be translated into SpecTRM-RL models, and then the SpecTRM-RL suite of analysis tools can be used to perform formal and informal safety analyses on the task model in isolation or integrated with the rest of the modeled system.

We had hoped to be able to apply these modeling languages and analysis tools to a TAP air/ground trajectory negotiation scenario, but the development of the tools took more time than we anticipated.

Designing Automation to Reduce Operator Errors

Nancy G. Leveson
Computer Science and Engineering
University of Washington

51-03

Everett Palmer
NASA Ames Research Center

Introduction

Advanced automation has been accompanied, particularly in aircraft, with a proliferation of modes, where modes define mutually exclusive sets of system behavior. The new mode-rich systems provide flexibility and enhanced capabilities, but they also increase the need for and difficulty of maintaining mode awareness. While automation has eliminated some types of operator mode-awareness errors, it has also created the potential for new types of mode-related problems [SW95].

After studying accidents and incidents in the new, highly automated aircraft, Sarter and Woods have concluded that certain errors are non-random and predictable [SW95]: They are the regular and predictable consequences of a variety of identifiable factors. Although these errors are accentuated by poor interface design and gaps or misconceptions in the user's mental model of the system, an important factor is inconsistent automation behavior.

Sarter and Woods have identified some of these predictable error forms. Leveson et. al. [LPS97] and Degani [Deg96] have defined taxonomies of automation features that lead to mode confusion. This paper describes an approach to dealing with mode confusion errors by first modeling blackbox software behavior and then using analysis methods and tools to assist in searching the models for predictable error forms, i.e., for features that contribute to operator mistakes. The analysis results can be used to redesign the automation, to change operator training and procedures, or to design appropriate human-computer interfaces to help avoid mistakes.

The approach requires a model of the blackbox behavior that is both formal and easily readable and reviewable by humans. The models we use are part of the software specifications in a methodology called SpecTRM (Specification Tools and Requirements Methodology) and thus the analysis is done directly on the system requirements specification and does not require extra modeling effort. SpecTRM includes a suite of analysis tools to detect errors and potentially hazardous behavior early in system development when tradeoffs and changes can more easily be made. In addition to providing design guidance, this approach might provide a way of "measuring" or evaluating the cognitive demands involved in working with specific automated devices. Hansman has suggested that automation complexity be defined in terms of the predictability of the automation behavior [Hans97]. This predictability can potentially be evaluated on the formal SpecTRM-RL (SpecTRM Requirements Language) models.

The rest of the paper provides more information about the approach and illustrates its use on a commonly reported mode confusion error called a "kill-the-capture" bust.

Mode Confusion Analysis

Most accidents related to software behavior can be traced back to errors or omissions in the software requirements, not to implementation or coding errors [Lev95, Lut93]. Although a great deal of effort has been expended in software engineering on finding software design and implementation errors, much less has been accomplished in terms of validating requirements specifications beyond executing them for a few test cases or showing the consistency of a formal specification with various properties of the underlying mathematical model [HL96, HLK95]. Most of the specfication errors and omissions that lead to accidents are unlikely to be found using these techniques. The testing of any complex software is necessarily very incomplete, and consistency with a mathematical model does not imply consistency with required properties of a real world application.

To deal with this problem, we have specified a set of criteria for completeness and correctness of blackbox process-control requirements specifications that are related to safety [JLHM91, Lev95]. These criteria were derived using real accidents and industrial experience with process-control software, and they have been validated by experimental application to the NASA Gallileo and Voyager software [Lut92, Lut93] and through industrial use. We are contining to extend the criteria, most recently with the goal of reducing mode confusion errors, and to validate them on real software [MLR97].

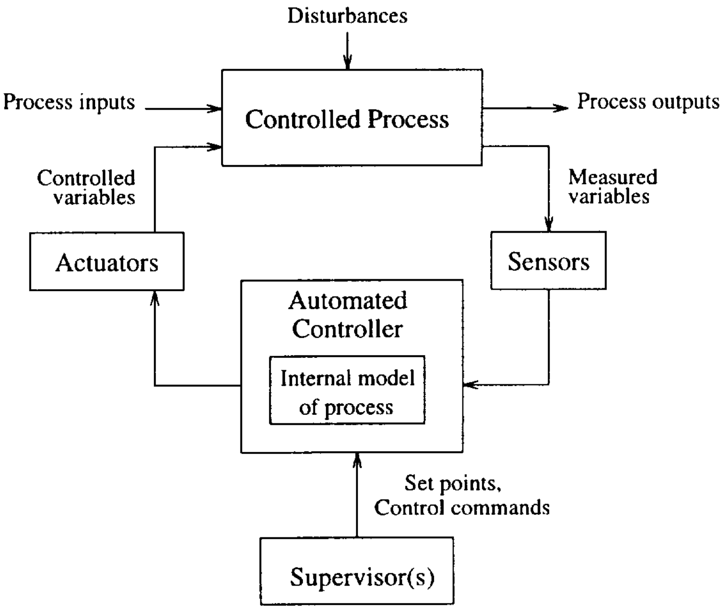


Figure 1: Simple Control Loop Model

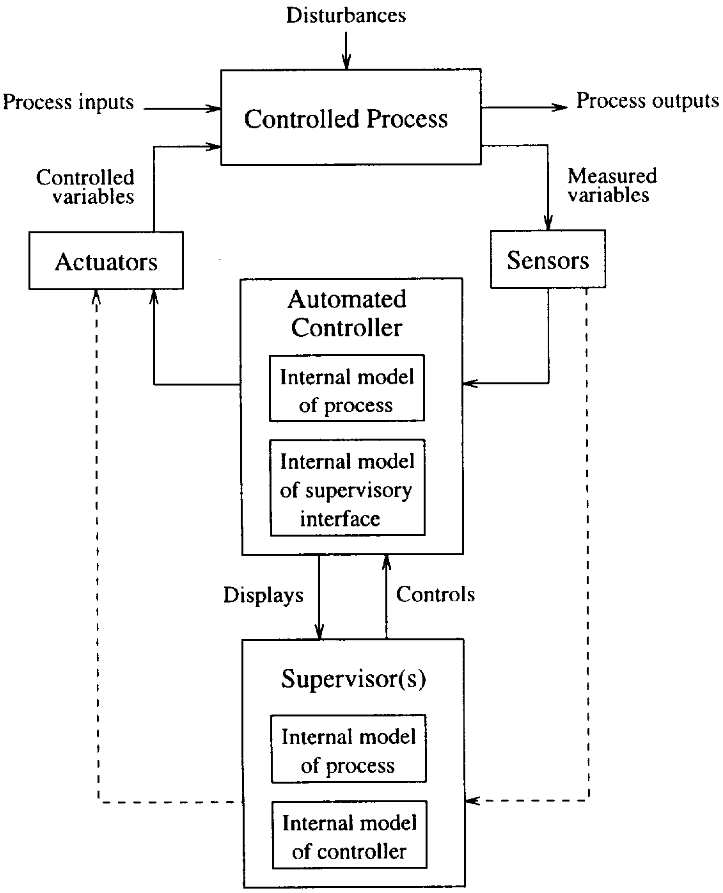


Figure 2: Modified Model to Account for Operator Error and Mode Confusion.

To apply the criteria, a blackbox state-machine model of the automation behavior is required. Blackbox requirements specifications do not contain information about internal design (the software design if the automated controller is a computer) but are written strictly in terms of externally visible inputs and outputs and the effects of these on a model of the process being controlled (see Figure 1). The process model is based on:

1. Current process state inferred from measured variables,
2. Past measured and inferred process states and variables,
3. Past outputs to actuators, and
4. Prediction of future states of the controlled process.

Accidents related to requirements (behavioral) specification occur when the internal model of the process becomes inconsistent with the state of the controlled process. This inconsistency may result from an incorrect model being specified originally (e.g., the model does not include basic required behavior for unusual or infrequently occurring cases) or from the modeled system state being updated incorrectly during execution, perhaps as a result of incorrect input from the sensors.

To define criteria related to mode confusion, we need to add a model of the controller-software interface to the automated controller. We also need to consider the supervisors' internal models of the expected behavior and state of the process and of the automated controller (see Figure 2). Accidents in this extended model may result from any of these models being incorrect or becoming inconsistent with the true state of the controlled process, the automated controller, or the supervisory interface (the human–computer interface). That is, accidents may result if any of the models are or become inconsistent with the state of the thing they are modeling and decisions or actions are made on the basis of the incorrect model. Criteria for correctness and safety can be specified in terms of these formal models and checked for particular system specifications.

Of course, we are not suggesting that it is possible to specify human mental models. Each person may have a different mental model of the system and the automation, and these may change over time within the same person. In fact, operators have been found to be able to function with multiple and inconsistent models [Luc87]. However, it is possible to state some high-level abstractions about required features of correct operator mental models—for example, that particular actions on the part of the operator will result eventually in particular changes in the automation and/or the system.

Note that we assume here that the operator's models are correct. This assumption will obviously not always be true. However, our approach involves first eliminating hazards for the ideal case. Then various types of hazard analysis can be used to determine which types of erroneous models will have the most serious consequences. The resulting information can be used for automation design, interface design, and operator training.

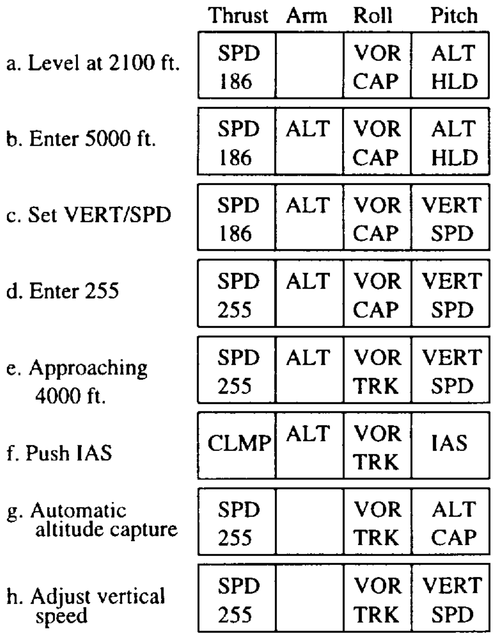
A previous paper described six categories of potential design flaws that can lead to mode confusion errors: interface interpretation flaws, inconsistent behavior, indirect mode changes, operator authority limits, unintended side effects, and lack of appropriate feedback [LPS97]. The rest of this paper shows an example of this approach for one particular common cause of mode confusion error, i.e., indirect mode changes. The basic criteria and analysis technique is being specified formally [Lev97], but we include only an informal description here.

Indirect Mode Change Example

Indirect mode changes occur when the automation changes mode without an explicit instruction by the operator. Such transitions may be triggered on conditions in the controller (such as preprogrammed envelope protection) or sensor input about the state of the controlled system (such as achievement of a target value). Indirect mode transitions create the potential for mode confusion and inadvertent activation of modes by the human controller. For example, the human controller may not update his or her models of the state of the process and the state of the automation and, based on these now incorrect models, issue an incorrect control command or fail to issue a required command.

An example of an accident that has been attributed to an indirect mode change occurred while an A-320 was landing in Bangalore. In this case, the pilot selection of a lower altitude while the automation was in the ALTITUDE ACQUISITION mode resulted in the activation of the OPEN DESCENT mode. It has been speculated that the pilots did not notice the mode annunciation because the indirect mode change occurred during approach when the pilots were busy and they were not expecting the change [SW95]. Another example of such an indirect mode change in the A-320 automation involves an automatic mode transition triggered when the airspeed exceeds a predefined limit. For example, if the pilot selects a very high vertical speed that results in the airspeed decreasing below a particular limit, the automation will change to the OPEN CLIMB mode, which allows the airplane to regain speed.

Palmer has described another example of a common indirect mode transition problem called a "kill-the-capture bust" that has been noted in many ASRS reports [Pal96]. Here we show the relevant parts of a SpecTRM-RL specification of the MD-88 control logic and describe how the problem can be detected and fixed.



	Thrust	Arm	Roll	Pitch
a. Level at 2100 ft.	SPD 186		VOR CAP	ALT HLD
b. Enter 5000 ft.	SPD 186	ALT	VOR CAP	ALT HLD
c. Set VERT/SPD	SPD 186	ALT	VOR CAP	VERT SPD
d. Enter 255	SPD 255	ALT	VOR CAP	VERT SPD
e. Approaching 4000 ft.	SPD 255	ALT	VOR TRK	VERT SPD
f. Push IAS	CLMP	ALT	VOR TRK	IAS
g. Automatic altitude capture	SPD 255		VOR TRK	ALT CAP
h. Adjust vertical speed	SPD 255		VOR TRK	VERT SPD

Figure 3: Flight Mode Annunciator (FMA) Displays for the Example Incident

In the incident, the crew had just made a missed approach and had climbed to and leveled at 2,100 feet. Figure 3 shows the sequence of Flight Mode Annunciator (FMA) values during the incident. The crew received the clearance to "...climb now and maintain 5,000 feet ...". The Captain set the desired altitude to 5,000 feet, set the autopilot pitch mode to vertical speed with a value of approximately 2,000 feet per minute and the autothrottle to SPD mode with a value of 256 knots (Figure 3(c) and (d)). Climbing through 3,500 feet, the Captain called for flaps up, and at 4,000 feet he called for slats retract and pushed the IAS button (Figure 3(f)). The pitch mode became IAS, and the autothrottle went to CLAMP mode. At this point, altitude capture was still armed. Three seconds later, the autopilot automatically switched to altitude capture mode. The arm window went blank, and the pitch window showed ALT CAP (Figure 3(g)). A tenth of a second later, the Captain adjusted the vertical speed wheel to a value of about 4,000 feet per minute. This speed adjustment caused the pitch autopilot mode to switch from altitude capture to vertical speed (Figure 3(h)). Climbing through 4,500 feet, the FMA was as shown in Figure 3(h), and the approaching altitude light was on. As the altitude passed through 5,000 feet at a vertical velocity of about 4,000 feet per minute, the Captain remarked, "Five thousand. Oops, it didn't arm." He pushed the ALT HOLD button and switched off the autothrottle. The aircraft continued to climb to about

5,500 feet and the ALTITUDE–ALTITUDE voice warning sounded repeatedly.

To identify and fix the problem, we use a formal model. A SpecTRM-RL model has two parts: a graphical model of the state machine and a specification of the logic on the transitions. Figure 4 shows part of the graphical SpecTRM-RL state machine model of the MD-88 vertical control logic needed to understand the incident and how to fix the software to avoid it. In order to keep the model small enough to fit in the paper, only parts of it are shown but during system engineering a complete model would be constructed. The graphical model has three main parts: the input–output interface (where the supervisory interface is one part), the operating modes of the automation itself (in this case the autoflight system), and the process model which includes both the process (aircraft) operating modes and models of the aircraft components.

In the supervisory interface, square boxes denote inputs and outputs having finite state values. Circles represent numbers. Note that this model represents the automated controller's view of the state of the interface, not necessarily the real state of the controls and displays. A complete safety analysis would evaluate if and how discrepancies between the two could occur and also whether such discrepancies could lead to hazardous system states.

The state transition logic is specified in SpecTRM-RL using a form of logic tables we call AND/OR tables. A transition can be taken if any of the columns of the table evaluates to *true*. A column evaluates to *true* if all the (non-blank) rows in a column are true. Figure 5 shows the relevant transition logic for the example.

The problem occurs because the transition to ALT CAP mode results in a transition of the capture mode to UNARMED before the altitude has actually been acquired. Although this is annunciated to the pilot by the Arm annunciator changing to blank when pitch mode changes to ALT CAP, the absence of an indicator is well known to be an error-prone way to notify the pilot of a mode change.

How could this be detected from examining the logic? In general, an indirect mode change is one that occurs without an explicit pilot action to change the mode. The vertical control logic for the example has three mode transitions that do not require direct pilot input: (1) the transition from ANY to ALT CAP, (2) the second column of the transition from ANY to ALT HOLD, i.e, when the altitude is acquired and the pitch is in mode ALT CAP, and (3) the second column of the transition from ARMED to NOT ARMED. Each of these mode transitions is triggered by a change in a controlled system variable or by internal mode change within the automation.

We will assume that the pilot's mental model includes a cause and effect relationship between arming the altitude capture and eventually (although it may not be immediately) acquiring that altitude and holding it:

set altitude and pull ALT → . . . → ALT HOLD.

Formal analysis will show that there is a path through the logic starting with the pilot pulling the altitude knob that does not result in the ALT HOLD state (specifically, this occurs when the automation is in the modes NOT ARMED and ALT CAP and the pilot does something that changes the pitch mode, in this case adjusting the vertical speed wheel).

One way to fix the problem is to change the transition logic to that shown in Figure 6. Note that although the second column of the transition table from ANY to ALT HOLD still does not require direct pilot input, the transition is not indirect by our definition because it satisfies the pilot model of the transition logic above. The transition from ANY to ALT CAP is still indirect, but there is no longer a path through the vertical control logic that violates the expected cause and effect relation between arming the capture and capturing the altitude when it is acquired. We note that this solution may violate other goals or desired behaviors of the autoflight system—the designers would have to determine this when deciding what solution to use. In addition, a more sophisticated solution may be required, e.g., a hysteresis factor may need to be added to the mode transition logic to avoid too rapid or "ping-ponging" transitions between pitch modes.

Finding indirect mode transitions does not mean the software must be changed. The identified criteria are simply clues for determining where to look for potential problems. The designers may decide that no real problem exists and make no changes or they may decide not to change the automation but instead to make changes in the interface design or in pilot training.

In general, it is not feasible to make all mode transitions direct in any sophisticated automated controller. The goal instead is to simplify the required pilot model of the automation behavior as much as possible. In this case, the pilot expects a direct mode transition from setting a target altitude and arming the altitude capture to eventually attaining capturing altitude, changing to ALT HOLD mode, canceling the ARM command. Any paths in the automation logic that will violate this assumption will be a source of potential mode confusion, even if the mode change is annunciated. Of course, the pilot may have more sophisticated knowledge of the automation logic and know that adjusting the vertical speed wheel will cancel the previously given altitude capture command. However, ,this knowledge assumes a much more

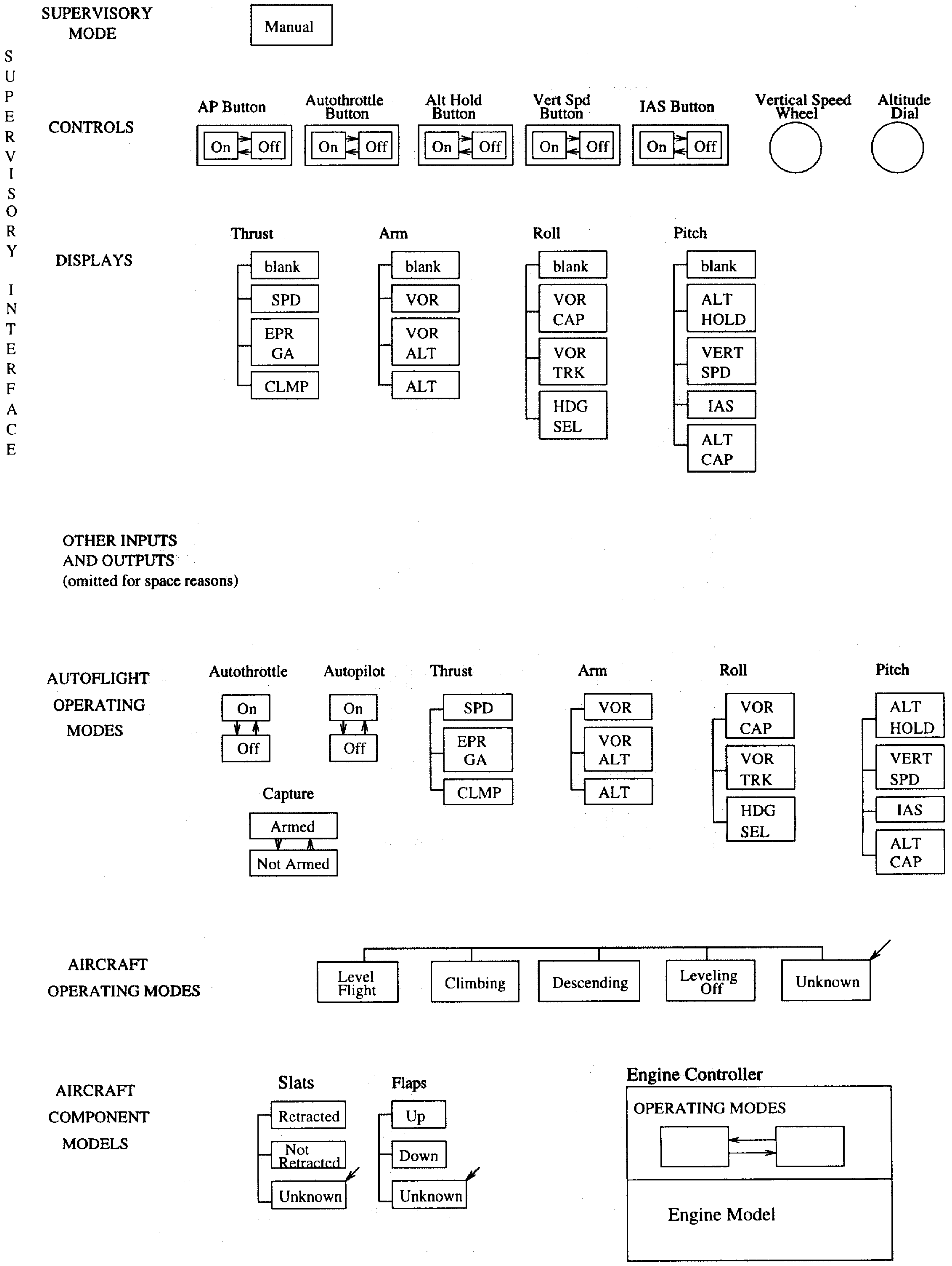


Figure 4: Part of the Graphical State Machine Model of the MD-88 Vertical Control Logic

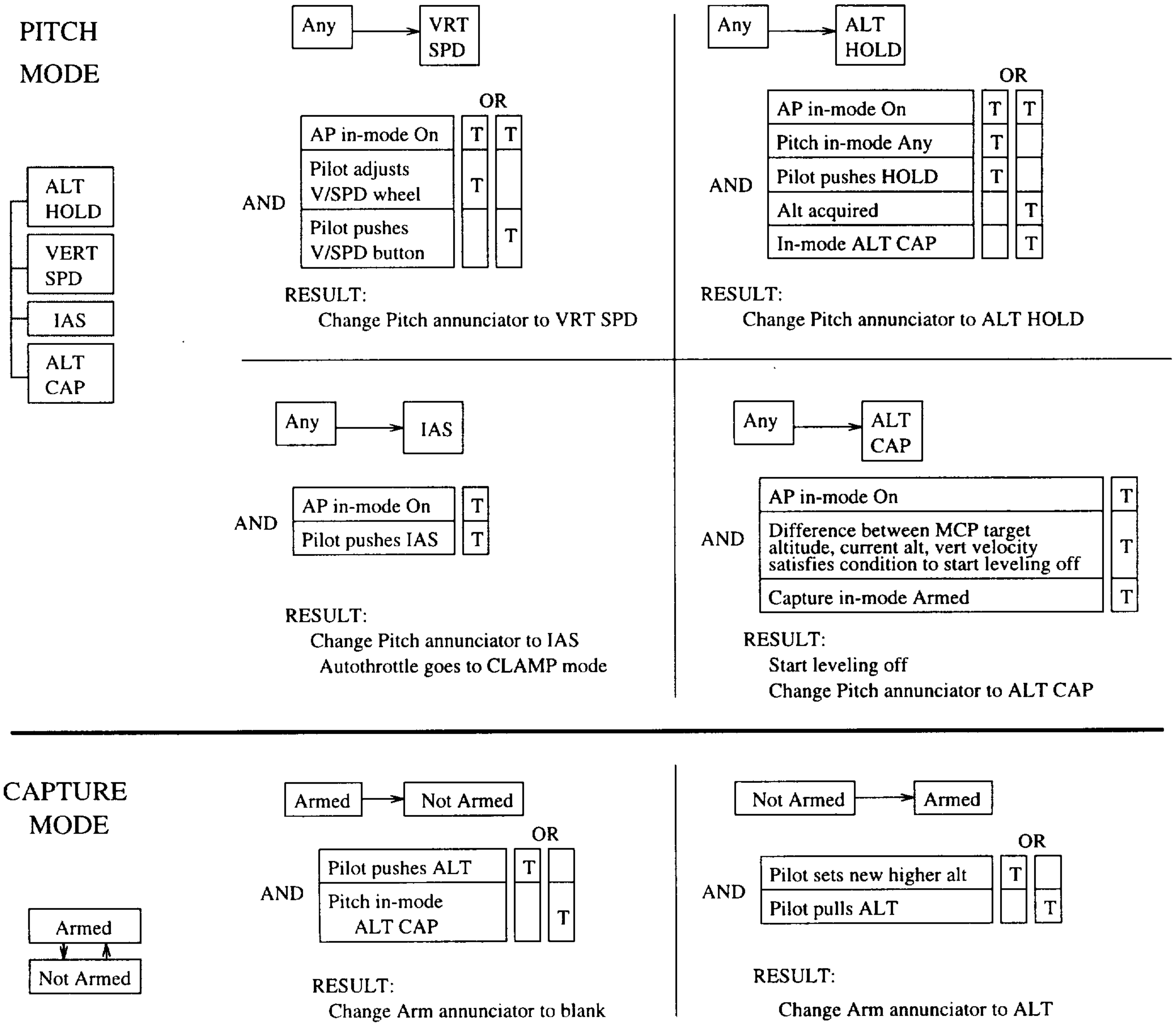


Figure 5: Transition Logic for the Pitch and Capture Modes

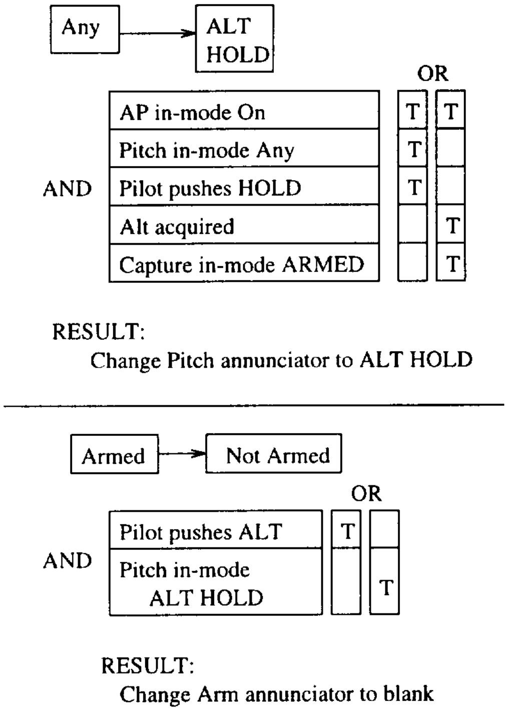


Figure 6: Revised Transition Logic for the Pitch and Capture Modes

complex model of automation behavior on the part of the pilot and makes the automation behavior more difficult to predict. The number of ASRS reports on this error leads us to believe that this assumption is not realistic.

Several caveats are important here. First, we are only guessing at the MC-88 software logic on the basis of the observed behavior. The real software specifications or code would have to be examined to determine what logic is actually implemented. Second, making the change we have recommended may not be feasible (or correct) due to other unmodeled parts of the logic that depend on the ARMED mode: A real development project would have the entire logic modeled and would be able to make the appropriate tradeoffs and design decisions.

REFERENCES

[Deg96] Degani, A. *Modeling Human-Machine Systems: On Modes, Error, and Patterns of Interaction.* Ph. D. thesis, Georgia Institute of Technology, 1996.

[Hans97] Hansman, John. Personal communication.

[HL96] Heimdahl, M. P. E. and N. Leveson. Completeness and consistency analysis of state-based requirements. *Transactions on Software Engineering*, June 1996.

[HLK95] Heitmeyer, C., Labaw, B., and Kiskis, D. Consistency checking of SCR-style requirements specifications. *Int. Symposium on Requirements Engineering*, York, 1995.

[JLHM91] 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.

[Lev95] Leveson, N.G. Safeware: System Safety and Computers. Addison-Wesley Publishing Co., 1995.

[Lev97] Leveson, Nancy G. Mode Confusion Modeling and Analysis, in preparation.

[LPS97] Leveson, N.G., Pinnell, L.D., Sandys, S.D., Koga, S., and Reese, J.D. Analyzing Software Specifications for Mode Confusion Potential. *Proc. Workshop on Human Error and System Development*, Glascow, March 1997.

[Luc87] Lucas, D.A. Mental models and new technology. *New Technology and Human Error*, John Wiley & Sons, 1987, pp. 337–340.

[Lut92] Lutz, R.R. Analyzing software requirements errors in safety-critical, embedded systems. *Software Requirements Conference*, 1992.

[Lut93] Lutz, R.R. Targeting safety-related errors during software requirements analysis. *Proc. Sigsoft '93: Foundations of Software Engineering*, 1993.

[MLR97] Modugno, F., Leveson, N.G., Reese, J.D., Partridge, K., and Sandys, S.D. Integrated Safety Analysis of Requirements Specifications. *Requirements Engineering Journal*, to appear.

[Pal96] Palmer, E. "oops, it didn't arm" – a case study of two automation surprises. NASA Technical Report, 1996.

[SW95] Sarter, N. D. and D. Woods "How in the world did I ever get into that mode?": Mode error and awareness in supervisory control. *Human Factors 37*, 5–19.

[SW95] Sarter, N. D. and D. Woods Strong, silent, and out-of-the-loop. CSEL Report 95-TR-01, Ohio State University, February 1995.

53-03

Modeling Controller Tasks for Safety Analysis*

Molly Brown and **Nancy G. Leveson**
Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
{molly,leveson}@cs.washington.edu

Abstract

As control systems become more complex, the use of automated control has increased. At the same time, the role of the human operator has changed from primary system controller to supervisor or monitor. Safe design of the human–computer interaction becomes more difficult.

In this paper, we present a visual task modeling language that can be used by system designers to model human–computer interactions. The visual models can be translated into SpecTRM-RL, a blackbox specification language for modeling the automated portion of the control system. The SpecTRM-RL suite of analysis tools allow the designer to perform formal and informal safety analyses on the task model in isolation or integrated with the rest of the modeled system.

1 Introduction

Increased complexity of control systems and advances in computer technology have combined to give automation a more authoritative role in control systems. As a result, many of these control systems rely on both human and automated controllers. For these controllers to interact effectively, the human–computer interaction must be carefully designed.

We began looking at these issues while working with the Terminal Area Productivity (TAP) Project at NASA Ames [PPC97]. The TAP Project is designing terminal area procedures for air traffic using data links in addition to voice contact to communicate trajectories and routing information between the air traffic controller and the aircraft.

Traditionally, HCI design has focused on the human user's point of view: what functionality is needed to support the tasks the human must accomplish. There have been numerous models for human task analysis developed to identify the knowledge and steps required to perform each human task.

Unlike traditional task modeling methods that focus on analyzing specific aspects of the HCI design, such as user goals or knowledge representation, our modeling technique has a more general focus. Our technique allows the designer to model the steps required to complete the task so she can study how these steps interact with the rest of the system. Specific environment cues required to complete a task are modeled as conditions on transitions between steps in a task. Depending on the visualization created to inspect the model, the analyst can focus on the knowledge needed throughout the task, on the steps required to complete the task or on some other model aspect of interest.

Our modeling methodology takes a system-centric view compared to the human-centered view of other task analysis methods. With a tighter coupling between the human and automated controllers in complex systems, the human controller must be viewed as a part of the entire system, therefore the interaction between the human and the computer should be viewed in the context of the entire system.

Our modeling technique focuses on formally analyzing a model of the controller's tasks independently and in the context of the complete control system model. Throughout the evolution of our method, we had three goals:

- To create a reasonable model of the actions of the human controller.
- To develop a model that can be formally analyzed with respect to safety concerns.
- To interface the model of the human controller tasks with formal models of the rest of the complex system.

*The research described has been funded by NASA Grant NAG-1-1894.

To realize these goals, we created a visual task modeling language that allows the analyst to easily represent the necessary information about the controller's tasks. These task models can be translated into the blackbox requirements specification language SpecTRM-RL and analyzed using a suite of analysis tools. SpecTRM-RL was developed to model all the components of control systems, therefore the SpecTRM-RL model of the controller tasks can be integrated with the model of the other components in the control system.

The rest of the paper is organized in the following way: Section 2 describes the SpecTRM tool suite and SpecTRM-RL modeling language; Section 3 describes our approach to achieving the goals we set for ourselves; Section 4 explains the example model that we use throughout the paper to illustrate our task modeling methodology; Sections 5 and 6 describe the visual task modeling language and how these models are created; Section 7 demonstrates how the safety analysis tools can be applied to these models; Section 8 discusses other work in the area of task modeling and analysis; and Section 9 discusses the contributions of this work and possible future questions to explore.

2 SpecTRM

The Software Safety Group at the University of Washington has developed a methodology for software specification called SpecTRM (Specification Tools and Requirements Methodology). Complex system development relies on multiple disciplines: system engineers, software engineers, human factors experts and application experts. SpecTRM takes a global system viewpoint to provide an environment to assist multidisciplinary teams. With a single consistent model of the complex system, analysts from each discipline can focus on the aspect of the system that is of interest to them.

The center of SpecTRM is SpecTRM-RL (SpecTRM Requirements Language), a formal requirements specification language for modeling blackbox behavior of control systems [Lev98]. SpecTRM-RL supports a wide-range of problem-solving strategies and tasks during system development and evolution.

Models written in SpecTRM-RL can be analyzed by the SpecTRM suite of tools. The SpecTRM tools were developed to detect errors and potentially hazardous behavior. The tool set is being expanded but currently includes:

- Model execution
- Automated formal analyses, such as consistency and completeness checks
- Automated tools to help with model exploration during forward and backward analyses
- Deviation analysis to test the robustness of system design to abnormal inputs
- Visualization tools to allow a wide-range of views of the executing model

These tools provide a flexible framework in which many complimentary analyses can be performed on a single model to help ensure total system safety.

3 Approach

We do not attempt to model erroneous human behavior, but limit our models to the expected controller behavior (both nominal and off-nominal) as defined in operational procedures. The models also include external inputs representing qualities of the environment that give rise to the human controller's decisions.

In many cases, the exact triggering conditions of a task are not necessary to have a meaningful model of the system. Any external or internal conditions that are particularly salient to the executing task are explicitly modeled while all other environmental effects are grouped into external conditions.

We quickly found that SpecTRM-RL models were not the most effective way to specify human tasks for system designers and human factors experts. For one thing, it was difficult to separate nominal from off-nominal behavior using the SpecTRM-RL notation. It was also difficult to see communication flow between the different components in the model. These limitations in expressibility led us to the development of a visual modeling language that has the characteristics lacking in SpecTRM-RL but remains easily translatable to SpecTRM-RL for analysis purposes.

4 Handoff Procedure Example

Throughout this paper, we will be using our model of a handoff procedure to illustrate our task modeling language. A "handoff", or a change of aircraft controller, occurs whenever an aircraft is changing from one controlling sector to another. The handoff procedure involves communication between the controller currently controlling the aircraft, the next controller to control the aircraft, and the pilot. Our model of this procedure includes the required tasks from each controller's point of view and the pilot's point of view. The handoff procedure model also includes a model of the radio used by the pilot to control the frequency to which she listens for controller communication. The radio component is included in the model because it is an integral part of the hardware/software portion of the system during the handoff procedure.

We chose this model to demonstrate our modeling and analysis technique because the model is complex enough to have interesting characteristics while remaining clear enough to be easily understood by readers who are not familiar with air traffic control. The handoff procedure consists of multiple components each with variety of tasks to complete and interesting interactions with other components.

5 Description of Visual Modeling Language

Figures 1, 2 and 3 show the visual model for current implementation of the handoff procedure in air traffic control systems.

The key on Figure 1 displays the components of the visual modeling language. *States* are used to represent each step required to complete a higher-level task. To change from one state to the next in the model, a *transition* must occur. Changing the current model state represents the completion of one subtask and the beginning of the next subtask. An *event* is the triggering condition for a transition and an *action* results from completing a transition. The default, or start, state is denoted by an arrow head on the left side of the state. For example, in the Current Controller model of Figure 1 the state *Aircraft being controlled by current controller* is the start state. The model can transition to the state *Initiating handoff* if the event *Conditions for handoff occur* fires. In this example, there is no resulting action from the transition.

Color is used in the model to differentiate between events and actions. The events triggering a transition are shown in blue text above the transition, while any actions resulting from the transition are shown in red text beneath the transition. A green outline around event or action text denotes that this is a communication point between two entities in the model. For example, in Figure 1 there is a green outline around the action *Initiate Handoff* in the Current Controller model and a green outline around the event *Initiate Handoff* in the Next Controller model. The green outline denotes that the action of initiating a handoff in the Current Controller model causes a transition to occur in the Next Controller model.

Task positioning is used to represent relationships among the steps in a task. The normative actions are seen on the main horizontal axis of each controller's task model. Any non-normative behavior diverges from the main axis until the situation has been corrected and the normative procedure resumes. For example, in the Next Controller model of Figure 1, rejecting the handoff is not the expected or nominal behavior of the Next Controller. The events, states and actions required to handle this sequence of subtasks are shown below the main axis of the Next Controller's behaviors. When the off-nominal steps have been completed, the model can return to a state on the main horizontal axis to represent returning to the nominal behavior.

The visual modeling language also allows the system designer to represent the relationship among the tasks that are being carried out by the model entities. To accurately represent the load on the human controller, the system designer must understand these relationships accurately. The system designer must understand which tasks are sequential and which tasks can be performed in any order. The Pilot model, Figure 2, shows how these task relationships are represented. The action *Issue frequency change* must occur before the Pilot can perform the subtasks to change the radio frequency and read back the Next Controller's frequency to the Current Controller. The branching in the transition arrows shows how changing the radio frequency and performing the read back are executed in some undetermined order. These tasks could be performed in parallel or in a sequential order chosen by the Pilot. The system designer must understand these possible interactions to ensure that the tasks do not overload the Pilot no matter what order she chooses to execute the tasks.

6 Construction of Task Models

The basis of the visual model is a task analysis. The task analysis identifies the major tasks of the controller components in the system then breaks these high level tasks into subtasks, down to the level of the key presses, voice communications, display cues, etc. involved in performing the task. From the task analysis, the visual model is created. The visual model easily represents the temporal relationships among the tasks carried out by a single component in the model and the relationships among multiple components in the model.

The SpecTRM-RL model is created based on the visual model. The conversion is a straightforward process of converting the visual relationships to a text-based modeling language. Whereas the visual modeling language uses entities, states, events and actions to represent the system, the SpecTRM-RL model decomposes the system into components, operating modes, and input/output interfaces. A *component* is a portion of the system with a well-defined interface to the rest of the system. Each high-level system entity and

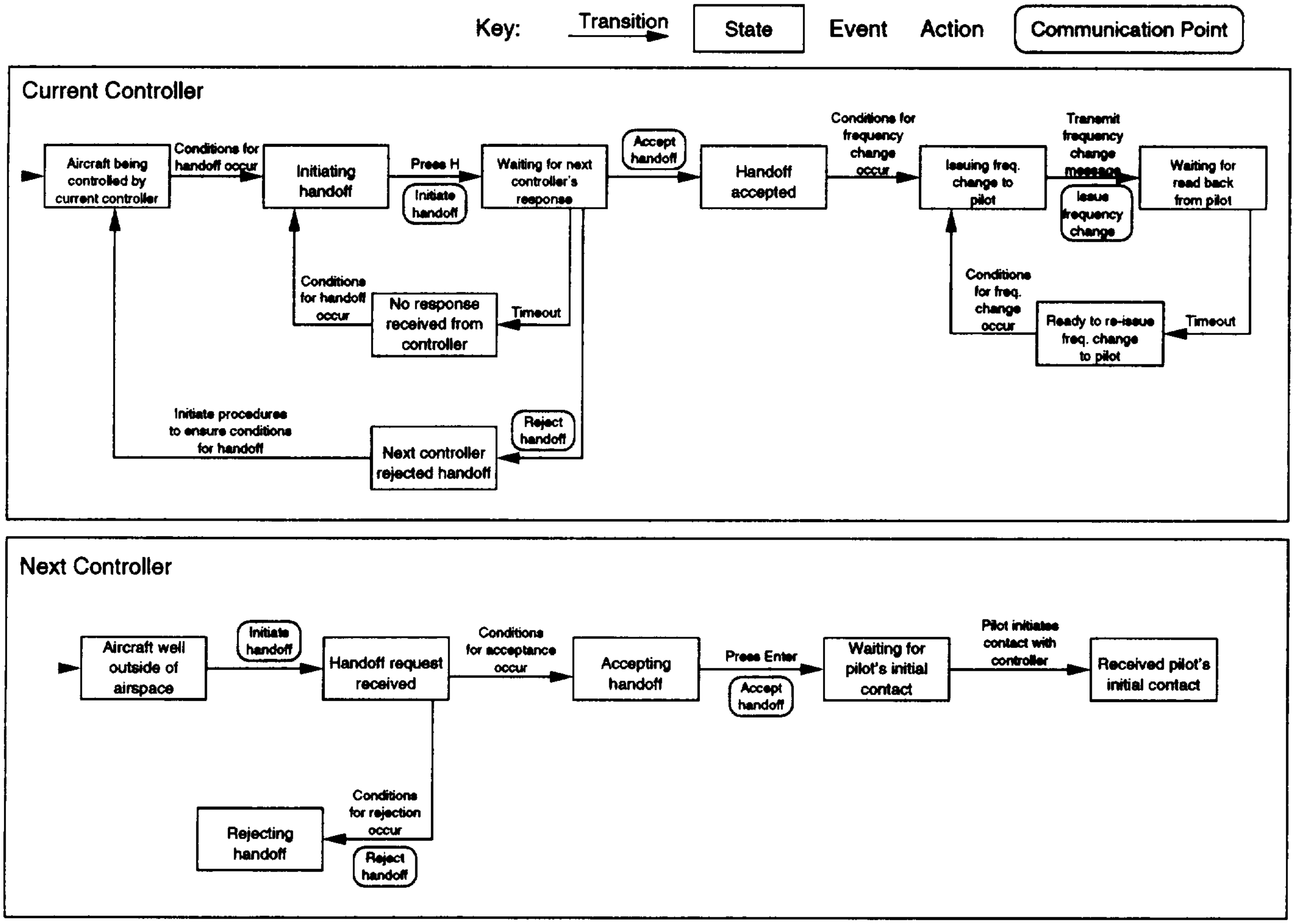

Key:
Transition
State
Event
Action
Communication Point
Current Controller
Aircraft being controlled by current controller
Conditions for handoff occur
Initiating handoff
Press H
Initiate handoff
Waiting for next controller's response
Accept handoff
Handoff accepted
Conditions for frequency change occur
Issuing freq. change to pilot
Transmit frequency change message
Issue frequency change
Waiting for read back from pilot
Conditions for handoff occur
No response received from controller
Timeout
Conditions for freq. change occur
Ready to re-issue freq. change to pilot
Timeout
Initiate procedures to ensure conditions for handoff
Reject handoff
Next controller rejected handoff
Next Controller
Aircraft well outside of airspace
Initiate handoff
Handoff request received
Conditions for acceptance occur
Accepting handoff
Press Enter
Accept handoff
Waiting for pilot's initial contact
Pilot initiates contact with controller
Received pilot's initial contact
Conditions for rejection occur
Reject handoff
Rejecting handoff

Figure 1: Model of Current Controller and Next Controller Tasks

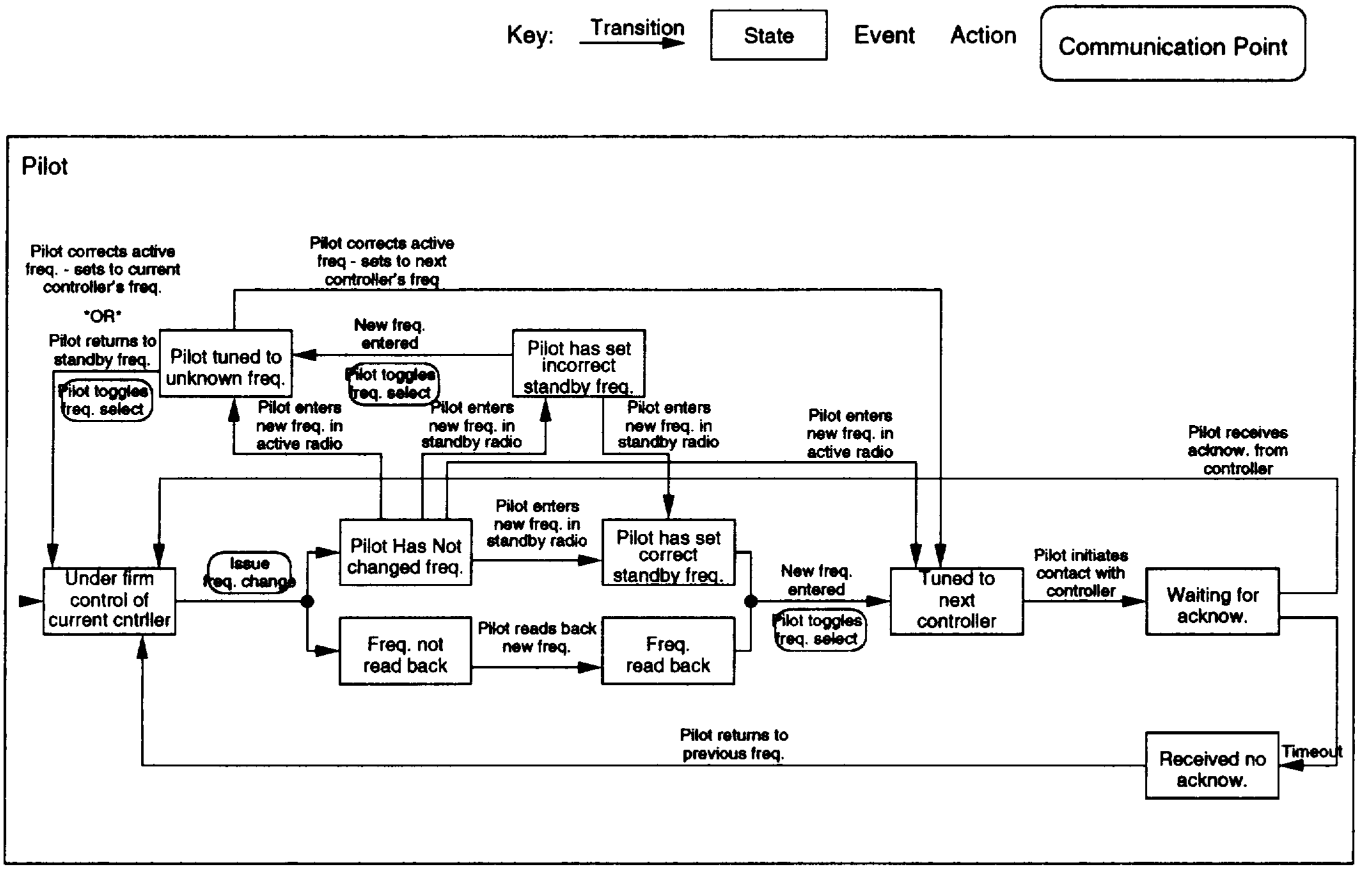

Key:
Transition
State
Event
Action
Communication Point
Pilot
Pilot corrects active freq. - sets to current controller's freq.
OR
Pilot returns to standby freq.
Pilot toggles freq. select
Pilot corrects active freq - sets to next controller's freq
Pilot tuned to unknown freq.
New freq. entered
Pilot toggles freq. select
Pilot has set incorrect standby freq.
Pilot enters new freq. in active radio
Pilot enters new freq. in standby radio
Pilot enters new freq. in standby radio
Pilot enters new freq. in active radio
Pilot receives acknow. from controller
Pilot enters new freq. in standby radio
Under firm control of current cntrller
Issue freq. change
Pilot Has Not changed freq.
Pilot has set correct standby freq.
New freq. entered
Pilot toggles freq. select
Tuned to next controller
Pilot initiates contact with controller
Waiting for acknow.
Freq. not read back
Pilot reads back new freq.
Freq. read back
Pilot returns to previous freq.
Received no acknow.
Timeout

Figure 2: Model of Pilot Tasks

the external environment become a component in the SpecTRM-RL model. For the handoff procedure, the components in the SpecTRM-RL model are the Current Controller, the Next Controller, the Pilot, and the Radio. The states in the visual model translate to *operating modes* in the SpecTRM-RL model. For example, from the visual model shown in Figure 3 the Radio component has two operating modes: *Frequency 1 Active/Frequency 2 Standby* and *Frequency 2 Active/Frequency 1 Standby.*

Translating the communication interfaces is slightly more difficult. The events in the visual model map to input interfaces for the SpecTRM-RL model components. If the event was marked as a communication point with another entity in the visual model (denoted by the green outline around the event label), the corresponding input will come from another SpecTRM-RL component. Otherwise, the event will map to an input from the external environment. Actions in the visual model translate to the output interfaces for the SpecTRM-RL components.

7 Analysis of Models

The human procedure model, along with SpecTRM-RL models of the other parts of the system, can be used in the safety analysis of the human-computer interaction. The SpecTRM tool set currently allows model execution, various types of safety analysis, and visualization.

7.1 Model Execution

Because the task models are executable, the system designer can inspect the specified dynamic interaction between the system components, including both the operators and the automated components. In this way, procedural errors, possible inconsistencies in the procedures, or incomplete procedural specifications can be detected. We found that model execution helped us to find several errors in the specification.

7.2 Safety Analysis Tools

The SpecTRM tool set has multiple fully automated or partially automated analyses that can be performed on the models to help identify possible unsafe aspects of the system requirements. Consistency and completeness analysis identifies inconsistencies in the specification and conditions not accounted for in the specification [HL96]. For example, the automated completeness check on this model found that the Pilot's behavior is not completely specified. During the handoff procedure, the model does not show how the Pilot should act if she reads back an incorrect frequency. This incompleteness in the specification is likely to be the result of an oversight of the system designer as she builds the model, but the completeness check aids the analyst by highlighting these possible oversights.

Deviation Analysis provides a way to evaluate the specification for robustness against incorrect inputs [RL97a, RL97b]. The analyst denotes potential hazardous outputs that she wants to check for and hypothesizes deviations in the inputs, for example, that measured speed is lower than actual speed. The Deviation Analysis tool will determine whether the deviation can lead to a hazardous state and, if not, whether the hypothesized deviation plus other conditions could lead to a hazard.

Our Backward Analysis tool allows the system designer to start from a hazardous state and work backward to determine if and how that state could be reached. Critical points in the design are identified that can be modified to avoid the hazardous state. For example, a backward analysis of the handoff procedure model found that the Pilot model can reach the hazardous state *Pilot Tuned To Unknown Frequency.*

7.3 Visualization Tools

Our IBToolKit (Interface Builder Tool Kit) allows the system designer to create visualizations that can be linked to a SpecTRM-RL model [Pin97]. In addition to simply showing the results of the model executing in a visual format, visualizations can be created that highlight specific system qualities. These types of visualizations can aid in the design of automated systems to maximize the strengths of both the human and automated controllers.

One example of a visualization that we created for the handoff procedure is shown in Figure 4. In this visualization, the cognitive demands on the pilot during task execution are highlighted as the model executes. The states *Pilot has not changed frequency* and *Frequency not read back* are highlighted in red to denote the current state of the model. From this visualization, the system designer can see that the pilot had to detect a change in her environment in order to transition out of the *Under firm control of current controller* state.

Another possible visualization would be a display of the cockpit. As the portions of the tasks are completed by the pilot interacting with the cockpit controls, the corresponding areas of the cockpit could be highlighted. This visualization assists the system designer in determining whether the actions involved in the task support the cognitive processes required of the pilot.

8 Related Work

Task analysis models such as Hierarchical Task Analysis (HTA) [AD67] focus on steps required by the human to complete the given task. In HTA, tasks can be broken down into subtasks and there is no restriction on the level of decomposition. The view is very human-centered since this model was initially proposed to represent tasks for the purpose of training new users of applications. The focus is on the steps that must be completed by the human to accomplish a task without consideration for the operations of the computer. HTA is very flexible, but this flexibility often leads to ambiguity and inconsistency among models.

Many task models have been based on the Goals, Operators, Methods and Selection Rules (GOMS) framework [CMN83]. In the GOMS methodology, the user's goals are decomposed into subgoals that have operators (behaviors) and methods (sequences of behaviors) associated with them. The goals are decomposed into four different levels of description. This model focuses on error-free performance and models the tasks down to a very detailed description of the keystroke level of behavior.

Task Analysis for Knowledge Descriptions (TAKD) takes a different approach to task modeling by focusing on the knowledge required by people to complete the tasks [DJ89]. The TAKD method allows the knowledge needed to complete a task to be represented by a multilevel abstraction that traces the high-level knowledge down to the low-level steps necessary to complete the task. Johnson et al. developed another task analysis approach, Task Knowledge Structures (TKS), based on TAKD [JJ91]. The TKS methodology represents tasks for the purpose of defining what tasks need to be supported by the system design. Unlike many other task models, TKS is a tool for designing systems, not just evaluating existing designs.

Executable task models have been developed and used for modeling human-computer interaction. Yost used the Soar/TAQL to model the Sisyphus-93 elevator-configuration task [Yos96]. To achieve an executable model, Yost began with a knowledge-level model of the task, refined this model to a problem-space model, then created the symbol-level model that is executable. The symbol-level model of the task was executed to help understand what problem-solving steps are involved during the completion of the task. The Programmable User Model (PUM), another executable model, provides a constrained cognitive architecture that can be programmed to simulate the user's execution of some set of tasks using a proposed interface [YGS89]. PUMs are meant to aid the interface designer in creating more usable designs by highlighting psychological considerations.

Joint human–computer controlled systems change the human controller's function from an active controller to a supervisor and error handler. However, automation design has not changed to support the human controller's new role as a monitor of system behavior and backup controller in case of emergency [SW95]. The Operator Function Model (OFM) provides a tool to help improve automation design so that the human controller tasks are supported [Mit96]. The OFM describes the required operator behavior by representing the operator functions, subfunctions and information needed by the operator for each activity.

9 Contributions and Future Work

We are in the process of testing this technique by modeling the controller tasks in the new air traffic control system being developed by the TAP Project at NASA Ames. We believe that the development of these tools will provide system designers and analysts with a more rigorous method of analyzing the assigned controller tasks for safety. The SpecTRM tool set provides automated analysis capabilities to the systems designer while the visual modeling language and flexibility of the IBToolKit allows the system designer to make design decisions with much better knowledge of the effect of these decisions on overall system safety.

There is future work to be done in the areas of special analyses on these types of models. We would like to see if we can use the models to assist in task allocation. Another problem we would like to explore using the models is mode confusion.

We would also like to continue exploring the flexibility in model analysis that is provided by the visualization tools. We are currently working on an interactive visualization of the ground procedures of the TAP FAST-system with datalink capabilities. This visualization to experiment with the use of an interactive model in system design.

References

[AD67] J. Annett and K.D. Duncan. Task analysis and training design. *Occupational Psychology*, 41:211–221, 1967.

[CMN83] S.K. Card, T.P. Moran, and A. Newell. *The psychology of human-computer interaction.* Lawrence Erlbaum Associates, 1983.

[DJ89] D. Diaper and P. Johnson. Task analysis for knowledge descriptions: theory and applications in training. In J. Long and A. Whitefield, editors, *Cognitive ergonomics and human-computer interaction.* Cambridge University Press, 1989.

[HL96] M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency analysis of state-based requirements. *Transactions on Software Engineering*, June 1996.

[JJ91] H. Johnson and P. Johnson. Task knowledge structures: psychological basis and integration in to system design. *Acta Psychologica*, 78(1-3):3–26, 1991.

[Lev98] N.G. Leveson. The SpecTRM-RL language. In preparation, 1998.

[Mit96] C.M. Mitchell. Task-analytic models of human operators: Designing operator-machine interaction. Technical report, Georgia Institute of Technology, 1996.

[Pin97] L.D. Pinnel. Visualizing requirement specifications: A toolkit for rapid prototyping of interfaces. Ph.D. qualifying project report, 1997.

[PPC97] T. Prevot, E. Palmer, and B. Crane. Flight crew support for automated negotiation of descent and arrival clearances. Technical report, NASA Ames Research Center, 1997.

[RL97a] J.D. Reese and N.G. Leveson. Software deviation analysis. In *International Conference on Software Engineering*, May 1997.

[RL97b] J.D. Reese and N.G. Leveson. Software deviation analysis: A safeware technique. In *AIChe 31st Annual Loss Prevention Symposium*, March 1997.

[SW95] N.D. Sarter and D. Woods. How in the world did I ever get into that mode?": Mode error and awareness in supervisory control. *Human Factors*, 37:5–19, 1995.

[YGS89] R.M. Young, T.R.G. Green, and T. Simon. Programmable user models for predictive evaluation of interface designs. In *Conference on Human Factors in Computing Systems (CHI '89)*, pages 15–19, May 1989.

[Yos96] G.R. Yost. Implementing the Sisyphus-93 task using Soar/TAQL. *International Journal of Human-Computer Studies*, 44(3-4), 1996.

S3.03

Analyzing Software Specifications for Mode Confusion Potential*

Nancy G. Leveson
L. Denise Pinnel
Sean David Sandys
Shuichi Koga
Jon Damon Reese

Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
{leveson,denisep,sds,skoga,jdreese}@cs.washington.edu

Abstract

Increased automation in complex systems has led to changes in the human controller's role and to new types of technology-induced human error. Attempts to mitigate these errors have primarily involved giving more authority to the automation, enhancing operator training, or changing the interface. While these responses may be reasonable under many circumstances, an alternative is to redesign the automation in ways that do not reduce necessary or desirable functionality or to change functionality where the tradeoffs are judged to be acceptable. This paper describes an approach to detecting error-prone automation features early in the development process while significant changes can still be made to the conceptual design of the system. The information about such error-prone features can also be useful in the design of the operator interface, operational procedures, or operator training.

Introduction

Today's large, complex systems often incorporate both human and automated control and monitoring. These jointly controlled systems are starting to experience accidents related to a lack of coordinated activity between the various controllers. One particularly problematic feature of these new designs is a proliferation of modes, where modes define mutually exclusive sets of system behavior.

The new mode-rich systems provide flexibility and enhanced capabilities, but they also increase the need for and difficulty of maintaining mode awareness, which can lead to new types of mode-related problems. This paper describes an approach to dealing with mode-confusion problems by analyzing the external blackbox behavior of the automation for potentially error-inducing features. The results can be used to make tradeoff decisions during the early development stages of the system.

While automation has eliminated some types of mode-awareness errors, it has also created the potential for new types of errors. Sarter and Woods extend the classic definition of mode error and distinguish between errors of commission (where an operator takes an inappropriate action) and errors of omission (where the operator fails to take a required action) [SW95].

The first automated systems tended to have only a small number of independent modes, and functions were associated with one overall mode setting. In addition, the consequences of operator mode awareness problems tended to be minor, partly because feedback about operator errors was fast and complete enough that operators were able to recover before the errors caused serious problems (Rasmussen's concept of *error tolerance*) [Ras90].

Studies of less complex aircraft automation show that pilots sometimes lose track of the automation behavior and experience difficulties with directing the automation, primarily in the context of highly dynamic and/or non-normal situations [SW95]. Sarter and Woods conclude that in most cases, these problems are associated with *errors of commission*, that

*The research described has been partly funded by NSF Grants CCR-9396181 and CCR-9520813 and NASA Grant NAG-1-1495.

is, with errors that require a pilot action in order for the problem to occur. This type of error is the classic mode error identified and defined by Norman—an intention is executed in a way that is appropriate for one mode but the device is actually in a different mode. Because the operator has taken an explicit action, he or she is likely to check that the intended effect of the action has actually occurred. The short feedback loops allow the operator to repair most errors before serious consequences result. This type of error is still the prevalent one on relatively simple devices such as word processors.

In contrast, studies of more advanced automation in aircraft like the A-320 find that mode *errors of omission* are the dominant form of error [SW95]. In this type of mode error, the operator *fails* to take an action that is required, perhaps because the automation has done something undesirable (perhaps involving a mode change) and the operator does not notice. In other words, the operator fails to detect and react to an undesired system behavior that he or she did not explicitly invoke. Because the mode or behavioral changes are not expected, the operator is less likely to pay attention to the relevant indications (such as mode annunciations) at the right time and detect the mode change or undesired behavior.

Errors of omission are closely related to the role change of the operator from direct control to monitor, exception handler, and supervisor of the automation. As these roles change, the operator tasks and cognitive demands are not necessarily reduced, but instead tend to change in their basic nature. The added or changed cognitive demands tend to congregate at high-tempo, high-criticality periods [SW95]. While some types of errors and failures have declined, new error forms and paths to system breakdown have been introduced.

Some of these new error forms are a result of mode proliferation without appropriate support. Providing support has been complicated by some unexpected changes in operator behavior in working with complex automation. For example, during long periods of flight, pilots do not have to monitor the mode annunciations continuously. Instead, they need to predict the occurrence of mode transitions in order to attend to the right indications at the right time. A-320 pilots have identified this new type of monitoring behavior in surveys conducted by Sarter and Woods. However, the automation and interfaces have been designed assuming conventional monitoring.

Simply calling for systems with fewer or less complex modes is unrealistic: Simplifying modes and automation behavior often requires tradeoffs with increased precision or efficiency and with marketing demands from a diverse set of customers [SW95]. However, systems may exhibit *accidental complexity* where the automation can be redesigned to reduce the potential for human error without sacrificing system capabilities. Where tradeoffs with desired goals are required to eliminate potential mode confusion errors, hazard analysis may be able to assist in providing the information necessary for appropriate decision making.

To identify and evaluate potential tradeoffs, we need to understand why the problems occur. Accidents in high-tech systems are related to complexity and coupling [Per84, Lev95]. Perrow distinguishes between accidents caused by component failures and those, which he calls *system accidents*, that are caused by interactive complexity in the presence of tight coupling. High-technology systems are often made up of networks of closely related subsystems (some of which may involve humans). Conditions leading to accidents emerge in the interfaces between subsystems and in their interactions, and coupling causes disturbances to progress from one component to another. Computers have exacerbated the problems by allowing new levels of complexity and coupling with more integrated, multi-loop control in systems containing large numbers of dynamically interacting components. Increased complexity and coupling make it difficult for the designer to consider all the system hazards, or even the most important ones, or for the operators to handle all normal and abnormal situations and disturbances safely.

Some of the increased complexity has been the result of what Sarter, Woods, and Billings have called *technology-centered automation* [SW95]. Too often, the designers of the automation focus on technical aspects and do not devote enough attention to the cognitive and other demands on the operator. Software engineers building embedded controllers are rarely taught or understand the set of cognitive processing activities associated with maintaining situation and mode awareness and how their designs can affect these human activities. Instead, they tend to focus on the mapping from software inputs to outputs, on mathematical models of required functionality, and on the technical details and problems internal to the computer. Little attention has been given to evaluating software in terms of whether it provides transparent and consistent behavior that supports operators in their monitoring and control tasks. In fact, the primary focus in software engineering and in artificial intelligence has been on producing automation that can function autonomously and not on supporting cooperation and communication between humans and computers.

The result of technology-centered automation has been what Wiener calls "clumsy automation." If it is true that mode-related problems are caused by clumsy or poorly designed automation, then changing the human interface, training, or operational procedures is not the obvious, or at least the only solution: "Training cannot and should not be the fix for bad design" [SW95]. Instead, if we can identify automation design characteristics that lead to mode awareness errors or that increase cognitive demands, then we may be able to redesign the automation without reducing system capabilities. In addition, knowing the causes of increased cognitive load will make changes in training or interface design more effective. The approach chosen will depend upon such factors as relative costs, perceived effectiveness, and required tradeoffs.

To accomplish this goal, designers need to be able to identify problematic design features. Our research goal is to identify design constraints on the automation based on known cognitive constraints on the human operator and engineered or natural environmental constraints. The first step in accomplishing this goal is to identify the types of errors that humans make in highly automated systems. Using this information, we can analyze the blackbox behavior specified in the automation requirements to predict where errors will occur and use this information to design the automation and the operator procedures, tasks, and interface. At first, we are simply going to analyze current designs, but our long term goal is to identify software design criteria and techniques that will help to create better designs from the beginning.

For our proposed analysis approach to work, human errors must be non-random. After studying accidents and incidents in the new, highly automated aircraft, Sarter and Woods have concluded that certain errors are predictable [SW95]: They are the regular and predictable consequences of a variety of identifiable factors. Although they are "accentuated" by poor interface design and gaps or misconceptions in the user's mental model of the system, mismatches between expected and actual automation behavior is not necessarily related to an inadequate operator mental model but can also result from inconsistent automation behavior. Sarter and Woods identify some of these error forms. Degani has also identified some features that lead to mode confusion [Deg96], and Jaffe [JL89, Lev95] has identified general requirements completeness criteria to eliminate some types of human–computer interaction errors.

We want to build on the work of Sarter and Woods, Degani, and Jaffe to find the factors or "predictable error forms" that relate to automation design and devise ways to identify these factors in software requirements specifications. Our approach is to model software blackbox behavior and provide analysis methods and tools to search the models for predictable error forms. In addition to providing design guidance, this approach might provide a way of "measuring" or evaluating the cognitive demands involved in working with specific automated devices. Hansman has suggested that automation complexity be defined in terms of the predictability of the automation behavior [Hans97]. This predictability can potentially be evaluated using our approach.

Analyzing designs requires an appropriate modeling and specification language. This language must be both formally analyzable and readable without advanced mathematical training. While automated tools may be necessary to analyze some aspects of large and complex models, we believe (and our empirical evidence supports the view) that the most important errors will be found by human experts [MLRPS97]. Therefore, one of our goals in the design of our modeling language and tools is to provide support in human navigation and understanding of complex models and specifications. In addition, any potential design flaws detected by automated tools will need to be evaluated by humans. Thus, readability of the models is also a requirement for human processing of the analysis results. Finally, the economics of system development are unlikely to allow for special formal models to be built. Instead, our analysis tools work directly on system and software requirements specifications.

In the following sections, we define the concept of a "mode" more carefully, describe our modeling language, describe criteria for detecting some types of mode ambiguity, and demonstrate how these criteria might be used in analyzing the blackbox behavior of the automation. The language and analysis are illustrated using a model of a NASA robot built to service tiles on the Space Shuttle.

Definition of a Mode

A *mode* defines a mutually exclusive set of system behaviors. One convenient way to describe behavior is to use state machine models. A machine or system can be thought of as having a set of states. The behavior of the system can be described by the possible transitions from one state to another. Those state transitions are triggered by events, conditions, or simply the passage of time (which can be thought of as an event). As an example, the following table shows the possible transitions between states given two system modes: startup mode and normal operation mode:

	a	b	c	d	e
Startup	c	b	d	e	a
Normal	c	d	a	e	a

Table 1: A simple state machine with two modes

The startup and normal processing modes in this machine determine how the machine will behave. For example, if the conditions occur that trigger a transition from state c, the machine will transfer to state d if it is in startup mode or to state a if it is in normal processing mode.

A basic tenet of linear control theory is that every controller contains a model of the general behavior and current state of the controlled system. This model may be embedded in the control logic of an automated controller or in the mental model of a human controller. The model is updated and kept consistent with the actual system state through various forms of feedback from the system to the controller. 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 [Lev95]. The situation becomes more complicated when there are multiple controllers because the models of the various controllers must also be kept consistent. A pilot, for example, must not only have a valid model of aircraft behavior but must also have a model of the automated systems' behavior in order to monitor or control the automation as well as the aircraft.

Mode confusion errors result from divergent controller models. See figure 1. Note that there are several sources of inconsistency due to improper feedback.

In attempting to categorize factors that predict mode errors, it is useful to distinguish between different types of modes. Degani classifies modes into three types [Deg96]:

1. *Interface modes* specify the behavior of the interface. They are used to increase the size of the input or output space.

2. *Functional modes* specify the behavior of the various functions of the machine.

3. *Supervisory modes* specify the level of interaction or supervision (manual, semi-automatic, or automatic).

We also define three types of modes, but classify them differently. The modes are defined with respect to the control component being specified:

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 (see Figure 1). In addition, each component may have multiple controllers (supervisors). For example, a flight guidance system may be issued direct commands by the pilot(s) or by another computer that is itself being supervised by the pilot(s). The robot motor controller (MAPS) described in the next section can be in either manual supervisory mode and controlled by a human operator, or it can accept control instructions from another computer called the "planner." 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 interface (Degani's interface modes) or to describe its required process-control behavior. For example, MAPS operation may be enabled or disabled at any time, depending on whether it is safe for MAPS to move the robot.

3. *Controlled-system operating modes* specify sets of related behaviors of the controlled system and are used to indicate its operational state. For example, the MAPS model of the robot indicates whether it is in a moving mode (between work areas), in a work mode (in a work area and servicing tiles, during which time the robot is not controlled by MAPS but by the planner) or is in an unknown mode (which means that MAPS does not know whether the robot is in moving mode or work mode).

The Modeling Language

Most software errors leading to accidents can be traced to incorrect or incomplete specifications rather than to incorrect implementations. While developing hazard analysis techniques, we have been trying to understand how to design specification languages that will facilitate analysis (by both humans and automated tools) of system and software requirements

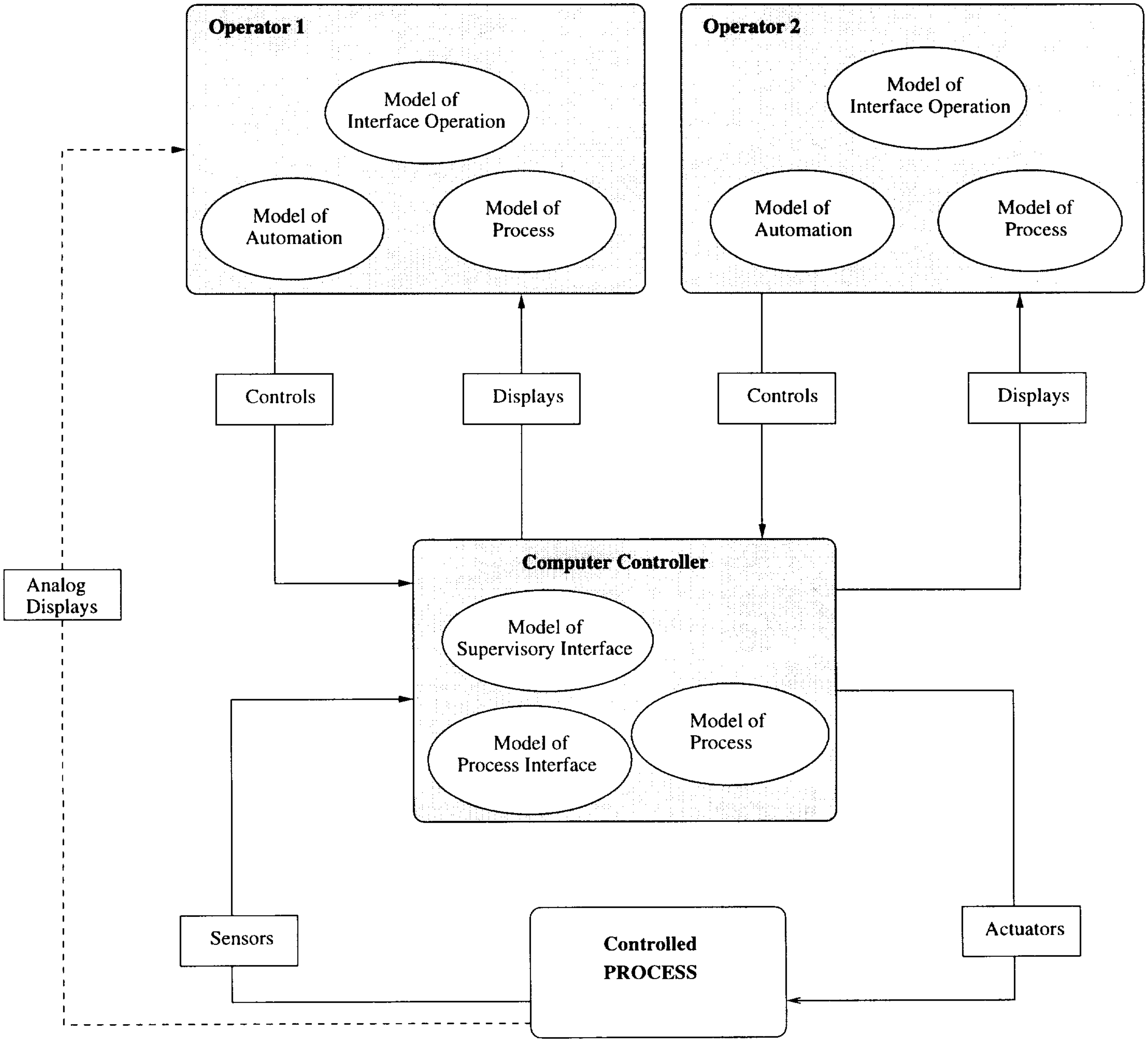


Figure 1: An example of a simple multiple-controller process-control system. To simplify the diagram, we have shown only one digital (computer) controller. In complex systems there may be several human, digital, or analog controllers at each level of hierarchical control and also more hierarchical levels than shown here. Note that each controller has several mental or logical models of the machine or process it is controlling as well as its interfaces. These models must be kept consistent for correct and safe monitoring and control.

specifications. We have found that effective error detection requires specifications that are readable and reviewable by human designers and application experts as well as analyzable by automated tools.

Our first language, RSML (Requirements State Machine Language), was designed while specifying the system requirements for TCAS II, an airborne aircraft collision avoidance system, for the FAA [LHHR94]. Using the lessons learned from this experience and others, we are designing a toolkit called SpecTRM (Specification Tools and Requirements Methodology) that includes a requirements specification language SpecTRM-RL. Underneath SpecTRM-RL there is a formal state machine model called RSM [JLHM91] upon which we have defined a set of correctness and completeness design criteria for safety-critical process-control system specifications.

One of our goals for SpecTRM-RL is to incorporate features to assist in designing less error-prone human-computer interactions and interfaces and in detecting potential communication problems, such as mode confusion. Although the notation has changed from RSML, the system behavior is still represented using hierarchical and orthogonal state machines. Because the majority of the errors and difficulty in reviewing our TCAS II model stemmed from the use of internal, broadcast events, we have eliminated this feature. We have also included features to assist in finding common dangerous omissions and errors in process-control requirements specifications. The language design is not quite complete, but Figure 2 shows part of an example specification for a NASA robot built to service tiles on the Space Shuttle. The software requirements are taken from a master's level project at the CMU Software Engineering Institute for part of a robot that was being designed and constructed in the Robotics Department [MMR92]. The system component used as the example in this paper is called MAPS (Mobility and Positioning Software).

Because our specifications are blackbox, they must describe the required behavior of the component (in this case MAPS) in terms only of inputs, outputs, the relationship between these, and a model of the controlled system. The specifications do not include any information about the implementation or internal design of the component, simply the input to output function it computes specified in terms of operating modes, an internal model of the controlled system, and an internal model of the interfaces with its supervisor(s) and the controlled process(es).

As with many complex control systems, this robot has multiple controllers and multiple levels of control. MAPS is a a mid-level controller responsible for issuing movement commands to the motor controller, which controls the mobile base of the robot (see Figure 3).

MAPS in turn can be controlled either by a human operator or by an onboard computer called the planner. The operator controls robot movement and positioning using a hand-held joystick. The planner can also control robot movement but does so by providing MAPS with a specification of the desired destination and route. Thus there are two supervisory modes: joystick and planner (see Figure 2). Either the human controller or the planner may assume control at any time, but the human controller is responsible for supervising the behavior of the robot at all times to prevent accidents, even when it is under planner control. Because of the distributed control structure, multiple possibilities for mode confusion exist.

The supervisory interface consists of the controls by which a supervisor directs the control component (in this case MAPS) and the displays by which the component relays information back to the supervisor. (Note that displays are not limited to visual displays; they can also include aural and other types of communication.) The operator can control MAPS using a joystick with two buttons and a keyboard, as shown in Figure 2. MAPS provides information to the operator via a graphical user interface. The MAPS behavioral requirements use only information about the content of the interface, not the specific layouts or design of the controls and displays (which is specified elsewhere). The communication interface with the planner is specified similarly.

In addition to the supervisory interface, there is an interface with the controlled system (other robot components), which includes the inputs and outputs between MAPS and the various sensors and actuators. These interface models are simply the view that MAPS has of the interfaces—the real interface(s) may contain different information due to various types of incorrect design or failures. By separating the assumed interface and the real interface, we are able to model and analyze the effects of various types of errors and failures.

The MAPS operational modes are:

- ENABLED or DISABLED: MAPS operation is enabled only if the safety circuit has signalled that the robot is in a safe state, the operator has depressed the deadman switch, and the robot's manipulator arm is stowed.
- OFF or OPERATIONAL: MAPS may be turned off or it may be operational.

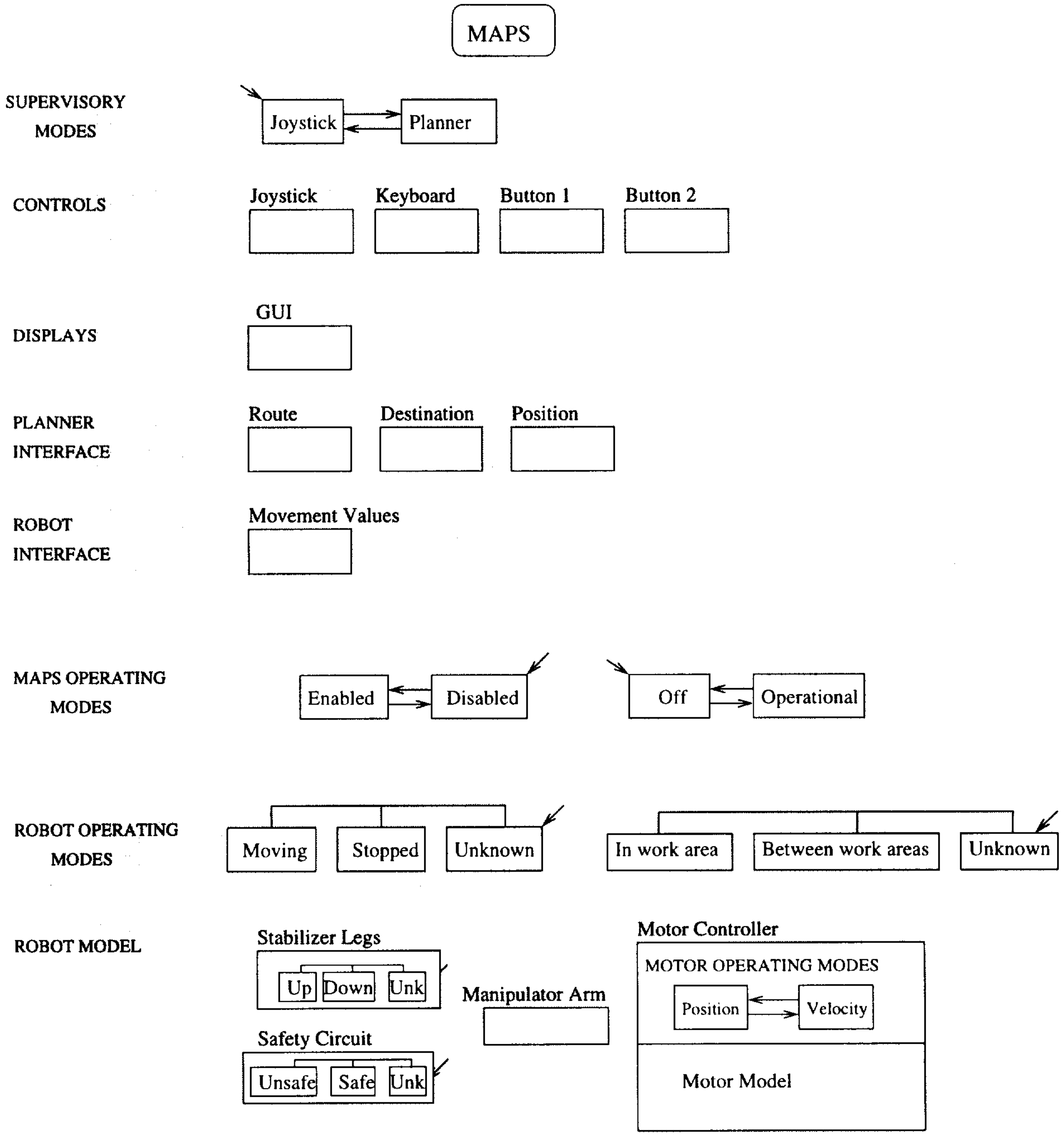


Figure 2: A partial specification of MAPS

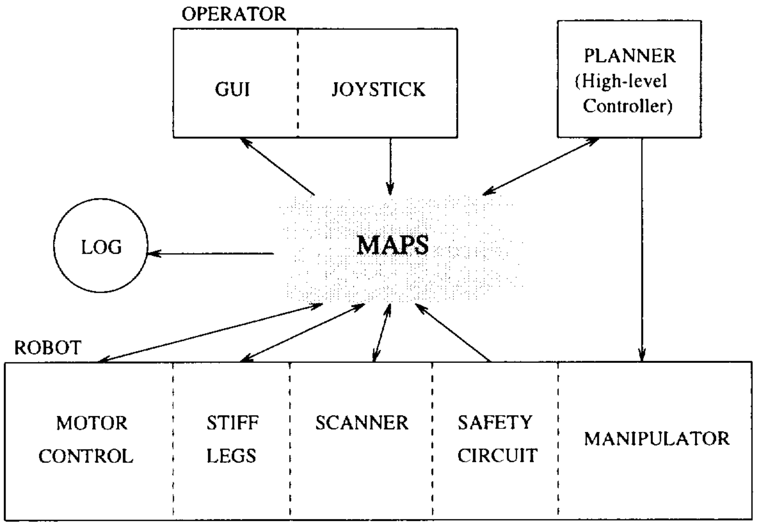


Figure 3: A high-level view of MAPS

The MAPS operational modes are relatively simple; a typical flight management component has a large number of such modes, leading to more potential for mode confusion.

The controlled system (in this case the robot) is described (within the MAPS model) in terms of its operating modes and a model of its relevant states. The robot controlled by MAPS is either MOVING or STOPPED (performing inspection and maintenance) or its operating mode is UNKNOWN. MAPS only controls motion; the servicing of the tiles is controlled by the planner. The robot is also either IN-A-WORK-AREA, BETWEEN-WORK-AREAS, or its location is UNKNOWN. The commands that MAPS issues will depend on these operating modes.

The last section of the MAPS high-level specification is the internal model of the state of the robot. Note that the interface and robot models are simply the internal models that MAPS has of the *assumed* state of the interface controls and displays and the *assumed* state of the robot, not their actual, physical state. The internal model may not be consistent with the real interface and robot states due to various types of errors and failures. For MAPS, the physical components of the robot that need to be modeled in order to specify the control algorithm are the stabilizer legs, the safety circuit, the manipulator arm, and the motor controller. Hierarchical control is common in complex systems: In this case, MAPS provides commands to the motor controller, which itself has operating modes and a state model.

The language enforces certain constraints to prevent design features that are known to lead to accidents. For example, in SpecTRM-RL, all components of the controlled system model (e.g., the robot model) must have an UNKNOWN state, which is the default for startup and for transitions from any type of temporary or partial shutdown to normal processing. The controlled system can usually continue to change state when the computer is shut down, and the software model of the process must be updated at startup or restart to reflect the actual process state. Many accidents have occurred in systems where the software assumed the status of the process had not changed since the computer was last operational and issued commands based on this erroneous information.

As with all state-machine models, transitions 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 use a form of logic table we call AND/OR tables. Figure 4 shows an example specification of a transition.

Once a blackbox model of the required system behavior has been built, this model can be evaluated as to whether it satisfies design criteria that are known to minimize errors and accidents.

Design Criteria and Analysis

Jaffe [JLHM91] originally identified 26 completeness criteria for requirements specifications, which we have now extended to close to 50 criteria [Lev95].

Planner → Joystick

Condition:

		OR	
A	ENTERED *Enabled*	T	·
N	*Operator Selects Joystick Mode*	·	T
D	*Safety Circuit* IN-STATE *Safe*	·	T
	Joystick IN-STATE *Neutral*	·	T

References:

Figure 4: An example of a transition definition in SpecTRM-RL. The supervisory mode transitions from PLANNER to JOYSTICK if the enabled state is entered or if the following three conditions are true: the operator selects JOYSTICK mode, the safety-circuit is in state SAFE, and the joystick is in the NEUTRAL position.

These criteria include a mixture of absolute criteria as well as heuristics for finding flaws that frequently lead to accidents. Many of these are related to human-computer interaction such as providing appropriate feedback during graceful degradation and completely specifying preemption logic when multi-step operator inputs can be interrupted before they are complete.

A few of the Jaffe criteria were derived from mathematical completeness aspects of the underlying formal RSM model, but most resulted from the experience Jaffe had in building such systems over a large number of years. These lessons learned were used to define design criteria for the formal RSM model. In attempting to extend the criteria (design constraints) to cover mode-confusion errors, we have taken the same approach.

Using the results of Sarter and Woods' studies of A-320 accidents and incidents along with other reports of mode-related error and building on the five types of mode-confusion design features identified by Degani, we have identified approximately fifteen design features of blackbox automation behavior not in our original Jaffe criteria that can lead to operator mode confusion or mode awareness errors. The work is in the preliminary stages, and the list will undoubtedly change as we investigate further. In this section, we illustrate the approach by describing a few items on the preliminary list and demonstrate their application to MAPS (where applicable).

Applying our criteria to a complex control system will almost surely identify a large number of behaviors that could lead to mode confusion. Getting rid of all such behaviors would most likely result in an overly simple control system that does not satisfy many of its goals. Instead, this information should be used to eliminate accidental complexity (i.e., the same functionality can be achieved but in a less error-inducing manner), to provide information for safety tradeoff analyses (perhaps by applying hazard analysis to the identified behaviors), and to design interfaces, operational procedures, and operator training programs. For example, accidents most often occur during transitions between normal and non-normal operating modes or while operating in non-normal modes. Therefore, the non-normal mode transitions should be identified and have more stringent design constraints applied to them.

The rest of the paper describes six of our categories of potential design flaws: interface interpretation errors, inconsistent behavior, indirect mode changes, operator authority limits, unintended side effects, and lack of appropriate feedback. Additional criteria can be found in [Lev95] and others will be described in future papers.

Interface Interpretation Errors

Interface mode errors are the classic form of mode confusion error: the computer interprets user-entered values differently than intended or it maps multiple conditions onto the same output depending on the active controller operational mode and the operator interprets the interface erroneously. The latter is Degani's *two plant state, one display* flaw.

A common example of an input interface interpretation error occurs with many word processors where the user may think they are in *insert* mode but instead are in *command* mode and their input is interpreted differently than they intended.

An example of an output interface mode problem was identified by Cook et.al. in a medical operating room device with two operating modes: warmup and normal [CPWM91]. The device starts in warmup mode when turned on and changes from normal mode to warmup mode whenever either of two particular settings are adjusted by the operator. The meaning of alarm messages and the effect of controls are different in these two modes, but neither the current device operating mode nor a change in mode are indicated to the operator. In addition, four distinct alarm-triggering conditions are mapped onto two alarm messages so that the same message has different meanings depending on the operating mode. In order to understand what internal condition triggered the message, the operator must infer which malfunction is being indicated by the alarm.

A more complex example occurs in a proposed A-320 accident scenario where the crew directed the automated system to fly in the TRACK/FLIGHT PATH ANGLE mode, which is a combined mode related to both lateral (TRACK) and vertical (FLIGHT PATH ANGLE) navigation:

> When they were given radar vectors by the air traffic controller, they may have switched from the TRACK to the HDG SEL mode to be able to enter the heading requested by the controller. However, pushing the button to change the lateral mode also automatically changes the vertical mode from FLIGHT PATH ANGLE to VERTICAL SPEED—the mode switch button affects both lateral and vertical navigation. When the pilots subsequently entered "33" to select the desired flight path angle of 3.3 degrees, the automation interpreted their input as a desired vertical speed of 3300 ft. This was not intended by the pilots who were not aware of the active "interface mode" and failed to detect the problem. As a consequence of the too steep descent, the airplane crashed into a mountain [SW95].

Several design constraints can assist in reducing interface interpretation errors. The first is that any mode used to control interpretation of the supervisory interface should be annunciated to the operator (that is, it should be part of the displays interface in our modeling language). More generally, the current operating mode of the automation should be annunciated (should be in the displays interface) as well as being part of the operating modes. In addition, any change of operating mode should trigger a change in the current operating mode reflected in the interface (and thus displayed to the operator), i.e., the annunciated mode must be consistent with the internal mode. Consistency between displayed and current mode is, of course, an obvious design constraint and a violation almost always signals an error in the requirements specification. The first constraint should hold for almost all systems as well.

Degani notes a third type of interface confusion error that results from mapping a single input control action to multiple internal mode changes, depending on the order of the control actions. He calls this *circular mode transitions.* For example, pushing a button on a device with a small input interface (e.g., a watch with one or two buttons) will often cycle through the possible modes, going to the next mode with the next button push. A possible design constraint here is that if a control input is used to trigger a mode transition, then it must be associated with only one mode change, that is, the mapping from control inputs to mode changes is one-to-one (a mathematical function). Note that it is unlikely that one would want to require that the function be bijective, because that would eliminate the possibility of all indirect mode changes. For some simple devices, even the constraint that the function be injective (one-to-one) may be impossible to enforce, and feedback about the current mode is the only possible solution to the problem.

Another design constraint related to these types of interface interpretation errors is that interpretation of the supervisory interface should not be conditioned on modes (an example is the accident related to the interpretation of "33" described earlier). This constraint is much stronger than the first three and may not always be feasible or desirable to enforce. However, our analysis tools will highlight these transitions to the designer/analyst so that appropriate scrutiny can be applied to that part of the design. Degani's circular mode transition is a subcase of this design constraint.

In the MAPS design, while MAPS movement is being supervised by the automated planner, the operator is removed from process control and acts simply as a safety monitor. There are six conditions under which MAPS will stop the movement of the robot: (1) the robot reaches the work area, (2) MAPS is disabled, (3) MAPS enters planner mode, (4) MAPS enters joystick mode, (5) the safety circuit detects an unsafe condition, or (5) the deadman is released (Figure 5. Three of these actions involves the operator directly—the selection of the joystick mode, selection of the planner mode, and the release of the deadman switch—and the operator will know why the robot was stopped. In a fourth case, the safety circuit signals an unsafe state and an error message is generated and sent to the operator interface to indicate why the robot stopped. But the operator cannot differentiate between the other two reasons, and MAPS can enter the DISABLED mode without indicating the reason to the operator. A straightforward solution is simply to provide additional status messages to the display.

Inconsistent Behavior

A more complex type of mode-confusion error, which is more often related to errors of omission than the interface errors mentioned above, is triggered by inconsistent behavior of the automation. Carroll and Olson define a consistent design as one where a similar task or goal is associated with similar or identical actions [CO88]. Consistent behavior makes it easier for the operator to learn how a system works, to build an appropriate mental model of the automation, and to anticipate system behavior.

An example of inconsistency was detected in an A-320 simulator study involving a go-around below 100 feet above ground level. Sarter and Woods found that pilots failed to anticipate and realize that the autothrust system did not arm when they selected TOGA (take off/go around) power under these conditions because it did so under all other circumstances where TOGA power is applied [SW95]. Another example of inconsistent automation behavior, which was implicated in an A-320 accident, involves a protection function that is provided in all automation configurations except the altitude acquisition mode in which the autopilot was operating.

Consistency is particularly important in high-tempo, highly dynamic phases of flight where pilots may have to rely on their automatic systems to work as expected without constant monitoring. Even in more low pressure situations, consistency (or predictability) is important in light of the evidence from pilot surveys that their normal monitoring behavior may change on advanced flight decks [SW95].

Pilots on conventional aircraft use a highly trained instrument scanning pattern of recurrently sampling a given set of basic flight parameters. In contrast, some A-320 pilots explained that they no longer have a scan anymore but allocate their attention within and across cockpit displays on the basis of expected behavior. Their monitoring objective is to verify expected automation states and behaviors. If the automation behavior is not consistent, mode errors of omission may occur where the pilot fails to intervene when necessary:

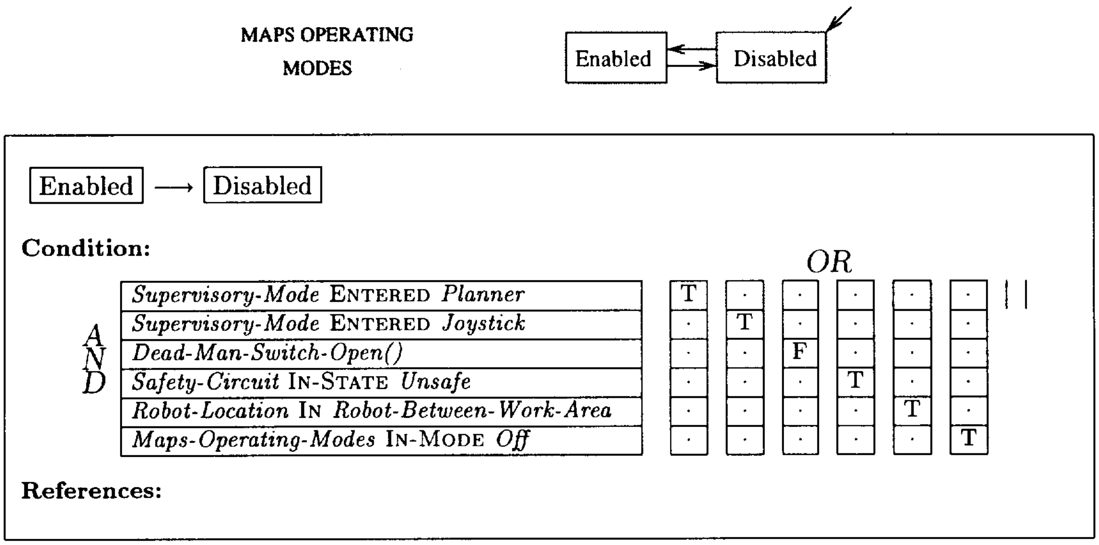
> Note the fundamental difference between these two monitoring strategies. In the case of a standard pattern, the pilot's attention allocation is externally guided while monitoring on advanced aircraft requires mental effort on the part of the pilot who has to determine on his own where to look next under varying task circumstances. Based on his expectations, the pilot only monitors part of all available data. Parameters that are not expected to change may be neglected for a long time. A standard instrument scan, on the other hand, serves to ensure that all relevant parameters concerning airplane behavior will be monitored at certain time intervals to make sure that no unexpected and maybe undesirable changes occur [SW95].

In our previous design criteria and analysis tools, we include a check for nondeterminism in the software behavior, that is, we check to determine whether more than one transition can be taken out of a state under the same conditions [HL96]. But consistency in this case requires more than simple deterministic behavior on the part of the automation. If the operator provides the same inputs but different outputs (behaviors) result for some reason other than what the operator has done (or even may know about), then the behavior is inconsistent from the operator viewpoint even though it is not mathematically inconsistent. More formally, inconsistent behavior results from two state transition functions of the form:

$$t_1 : s \times i_o \times x \rightarrow \{s \times O\}'$$
$$t_2 : s \times i_o \times y \rightarrow \{s \times O\}''$$

where $s \in \Sigma$ is a state, i_o is an operator input, O is an output, and x and y can be states, reference values, supervisory interface values, etc.

We have identified several different design constraints related to various types of inconsistency. However, there may be reasons why having such inconsistencies is necessary or reasonable. Again, our tools can point out such potential problems to the designer/analyst who must make the final decision about whether the automation should be changed. Because consistency may be most important during critical situations or when the behavior is related to a safety design constraint, our hazard analysis tools may be able to assist with these decisions and our new intent specifications [Lev97] (a form of Rasmussen's means-ends hierarchy adapted for software) can be used to



Enabled ⟶ Disabled

Condition:

AND		OR					
A	*Supervisory-Mode* ENTERED *Planner*	T	·	·	·	·	·
N	*Supervisory-Mode* ENTERED *Joystick*	·	T	·	·	·	·
D	*Dead-Man-Switch-Open()*	·	·	F	·	·	·
	Safety-Circuit IN-STATE *Unsafe*	·	·	·	T	·	·
	Robot-Location IN *Robot-Between-Work-Area*	·	·	·	·	T	·
	Maps-Operating-Modes IN-MODE *Off*	·	·	·	·	·	T

References:

Figure 5: Example of the conditions under which robot movement is stopped

trace such behavior back to its original system goals and safety constraints to identify any reasons for the specified inconsistent behavior.

Indirect Mode Changes

Indirect mode changes occur when the automation changes mode without an explicit instruction by the operator. Such transitions may be triggered on conditions in the controller (such as preprogrammed envelope protection) or sensor input about the state of the controlled system (such as achievement of a preprogrammed target or an armed state with a preselected mode transition).

Like many of the other mode-confusion problems noted in this paper, indirect mode transitions create the potential for mode errors of omission and of inadvertent activation of modes by the operator. Again, the problems are related to changes in scanning methods and difficulty in forming expectations of uncommanded or externally triggered behavior.

Behavioral expectations are formed based on the operators' knowledge of input to the automation and on his or her mental model of the automation's designed behavior. Gaps or misconceptions in the operator's mental model may interfere with predicting and tracking indirect mode transitions or with understanding the interactions between different modes.

An example of an accident that has been attributed to an indirect mode change occurred while an A-320 was landing in Bangalore. In this case, the pilot selection of a lower altitude while the automation was in the ALTITUDE ACQUISITION mode resulted in the activation of the OPEN DESCENT mode. It has been speculated that the pilots did not notice the mode annunciation because the indirect mode change occurred during approach when the pilots were busy and they were not expecting the change [SW95]. Another example of such an indirect mode change in the A-320 automation involves an automatic mode transition triggered when the airspeed exceeds a predefined limit. For example, if the pilot selects a very high vertical speed that results in the airspeed decreasing below a particular limit, the automation will change to the OPEN CLIMB mode, which allows the airplane to regain speed. As a final example, Palmer has described an example of a common indirect mode transition problem called a "kill-the-capture bust" that has been noted in hundreds of ASRS reports [Pal96]. Leveson and Palmer have modeled an example of this problem in SpecTRM-RL and shown how it could be detected and fixed [LP97].

Another example of indirect mode change can be found in the MAPS specification. In this scenario, MAPS is in joystick supervisory mode and it receives a message from the planner that the robot has reached the work area. This message will cause MAPS to transition from ENABLED to DISABLED mode (see Figure 5) without any explicit instruction from the human operator and without informing the operator of the mode change. If the analyst decides that this is a potentially dangerous scenario, the problem can be solved by augmenting the transition ANY → IN-WORK-AREA as seen in Figure 6.

In general, there are four ways to trigger a mode change:

1. Operator explicitly selects a new mode.
2. Operator enters data (such as a target altitude) or a command that leads to a mode change:
   (a) Under all conditions.
   (b) When the automation is in a particular state.
   (c) When the controlled system model or environment is in a particular state.
3. Operator does not do anything but the transition is triggered by conditions in the controlled system.
4. Operator selects a mode change but the automation does something else, either because of the state of the automation and/or the state of the controlled system.

The formal definitions are obvious and are omitted here. Degani also notes these types of indirect mode changes, but he gives them different names and classifies them differently than we do.

Operator errors associated with indirect mode changes are a phenomenon found primarily in advanced automation. Early automation tended to involve only a small number of independent modes. Most functions were associated with only one overall mode setting. We probably do not want to go back to automation that will change mode only in response to direct operator input, but design constraints are desirable that limit such indirect transitions and eliminate it when possible. Our analysis methods highlight mode changes that are independent of direct and immediate instructions from human supervisors, and our tools may also be able to assist the analyst in identifying the most hazardous indirect mode changes.

Operator Authority Limits

Interlocks and lockouts are often used to ensure safety. *Interlocks* are commonly used to prevent hazardous system states by enforcing correct sequencing of events or actions or to isolate two events in time. A *lockout* makes it impossible or difficult to enter a hazardous state.

Authority limiting is a type of lockout or interlock that prevents actions that could cause the system to enter a hazardous state. Such authority limitations must be carefully analyzed to make sure they do not prohibit maneuvers that may be needed in extreme situations. Recent events have involved pilots "fighting" with the automation over control of the aircraft after observing unexpected or undesirable aircraft or automation behavior.

Various types of authority limits are used to prevent operator error or to provide protection when the operator cannot or does not take proper action. For example, automation on advanced aircraft often has the ability to detect and prevent or recover from predefined unsafe aircraft configurations such as a stall. Once a hazardous state is detected, the automation has the power to override or limit pilot input.

Some accidents and incidents in highly automated aircraft have involved pilots not being able to overcome the protection limits or the pilots not being aware that the protection functions were in force. For example, the pilots during one A-320 approach disconnected the autopilot while leaving the flight directors and the autothrust system engaged. Under these conditions, the automation provides automatic speed protection by preventing the aircraft from exceeding upper and lower airspeed limits:

> At some point during the approach, after flaps 20 had been selected, the aircraft exceeded the upper airspeed limit for that configuration by 2 kts. As a consequence, the automation intervened by pitching the airplane up to reduce airspeed back to 195 kts. The pilots, who were not aware that the automatic speed protection was active, observed the uncommanded automation behavior. Concerned about the unexpected reduction in airspeed at this critical phase of flight, they rapidly increased thrust to counterbalance the automation. As a consequence of this sudden burst of power, the airplane pitched up to about 50 degrees, entered a sharp left bank, and went into a dive. The pilots eventually disengaged the autothrust system and its associated protection function and regained control of the aircraft [SW95].

Various design criteria are related to authority limits. For example, information about any modes or states where the operator input is ignored or limited must be provided in the supervisory interface. In addition, the analysis tools can examine the specified software behavior and detect exceptions to following

ANY → In-Work-Area

Condition:

AND	*Receive Planner-At-Work-Area Message*	T
	Supervisory-Mode IN-MODE *Planner*	T

References:

Figure 6: Modified transition to the IN-WORK-AREA mode.

operator requests. Again, the information in the intent specification is useful in determining whether such design features are intentional and whether they are related to identified hazards.

Unintended Side Effects

Mode ambiguity can also arise when an action intended to have one particular effect has an additional effect, i.e. an unintended side effect. An example occurred in the Sarter and Woods A-320 simulator study where it was discovered that pilots were not aware that entering a runway change *after* entering data for the assigned approach results in the deletion of all previously entered altitude and speed constraints even though they may still apply.

This type of design flaw differs from indirect mode changes in that the unintended change is not in the mode but in some other type of information, such as reference values. Degani describes this type of problem in terms of a mode/reference value interaction, but more generally the same problem occurs when any operator entry (for example, an input value rather than a mode change) has unintended side effects.

Unintended side effects can contribute to mode confusion, and often need to be evaluated by the design team. If a decision is made to keep the behavior, proper feedback constraints may be required to prevent the type of confusion that seems to result.

Lack of Appropriate Feedback

Many of the original Jaffe criteria or the newly defined criteria mentioned above are related to providing appropriate feedback (e.g., providing feedback about the status of interlocks and lockouts and providing graceful degradation). In general, operators need to have the information necessary to understand the mode transitions taken, i.e., the conditions that trigger transitions. Operators need not only to track the current active modes and to understand their implications, but they also need to keep track of other automation and system status information that may result in the indirect activation of modes. The difference between these design constraints and those requiring mode transition annunciations described in the section on interface interpretation errors is that in this case the automated system must not simply notify the operator that a mode change has already occurred (annunciate the present mode), but it must provide the information necessary for the operator to *predict* or *anticipate* mode changes.

Incomplete feedback is often implicated in accident scenarios. For example, in the A-320 Bangalore accident, the pilot flying (PF) had disengaged his flight director during the approach and was assuming that the pilot-not-flying (PNF) would do the same thing [SW95]. The result would have been a mode configuration in which airspeed is automatically controlled by the autothrottle (the SPEED mode), which is the recommended procedure for the approach phase. However, the PNF never turned off his flight director, and the OPEN DESCENT mode became active when a lower altitude was selected. This indirect mode change (explained above) led to the hazardous state and eventually the accident. But a complicating factor was that each pilot only received an indication of the status of his own flight director and not all the information necessary to determine whether the desired mode would be engaged. The lack of feedback or knowledge of the complete system state contributed to the pilots not detecting the unsafe state in time to reverse it.

Where automation has the ability to take autonomous actions (i.e., those not directly commanded by the operator), information interchange becomes crucial in coordinating activities and in detecting mismatches between expected and actual system behavior. A behavioral description of the software, as provided in SpecTRM-RL, is useful in determining exactly what information the operator needs to monitor and control the automated system.

The problems of providing salient feedback are, of course, much more complicated than simply identifying the information that needs to be conveyed, but identification is an important step in the process. In our original Jaffe criteria, we identified design constraints on basic feedback to the computer about the state of the controlled process and some types of operator feedback requirements, but these need to be augmented with a complete set of requirements on the feedback to the operator or automation supervisor. An example constraint is that operators must have access to all information on critical mode transitions in order to predict and monitor those transitions.

One important aspect of using feedback for error detection is the need for independent information. Errors can only be found through discrepancies in redundant information. One way to detect that automated equipment is not operating correctly is for operators to detect a discrepancy between the automation behavior and their mental model of how they think the automation should work. However, operators often have limited understanding of complex automation behavior or are afraid to step in.

In addition, often an error is only detectable using some information about the state of the environment or the controlled process. However, if the erroneous behavior is occurring because the automation is confused about the environment or system state, then it obviously cannot provide this information to the operator. That is, the automation may show only consistent information because it does not know there is an error in its system model. Therefore, it is not surprising that Sarter and Woods found that pilots mostly found errors through information given in nonautomated displays and instruments (i.e., based on observations between desired and actual aircraft behavior, not on indications of the nominal status of the automated systems). The same phenomenon is true for other types of systems. The problem is complicated by the fact that operators cannot always see what the automation is doing and can only tell by directly observing the reaction of the system or by getting feedback from some independent display. Providing independent feedback and providing more feedback on what the automation is doing can alleviate these problems.

Conclusions and Future Work

We have outlined an approach to reducing potential mode confusion errors. The software requirements are modeled using a hierarchical state machine language and then analyzed (manually or with automated assistance) to identify violations of a set of design constraints associated with mode-confusion errors. The approach was illustrated with a model of the software controlling a NASA robot and a description of a few of our currently identified software design constraints.

This work is still in the preliminary stages. We need to complete and partially validate our set of constraints by examining more accidents and incidents to determine whether the current set would identify the factors involved. Once we are fairly confident about our list, we plan to validate the feasibility of applying the constraints to real specifications by building a prototype analysis tool and applying it to a model of an advanced aircraft FMS (probably not the A-320, from which many of the constraints were originally derived). A possible step after that would be to use incident reports from one or more of the reporting systems (e.g, ASRS, CHIRP, or EUCARE) for that aircraft to see if our predictions are accurate.

References

[CO88] Carroll, J.M. and Olson, J.R. Mental models in human–computer interaction. in M. Helander (Ed.) *Handbook of Human–Computer Interaction*, Elsevier Science Publishers, pp. 45–65, 1988.

[CPWM91] Cook, R.I., Potter, S.S., Woods, D.D. and McDonald, J.M. Evaluating the human engineering of microprocessor-controlled operating room devices. *Journal of Clinical Monitoring*, 7, pp. 217–226, 1991.

[Deg96] Degani, A. *Modeling Human-Machine Systems: On Modes, Error, and Patterns of Interaction.* Ph. D. thesis, Georgia Institute of Technology, 1996.

[Hans97] Hansman, John. Personal communication.

[HL96] Heimdahl, M. P. E. and N. Leveson. Completeness and consistency analysis of state-based requirements. *Transactions on Software Engineering*, June 1996.

[JLHM91] Jaffe, M.S, Leveson, N.G., Heimdahl, M.P.E., and Melhart, B.E.. Software requirements analysis for real-time process-control systems. *IEEE Transations on Soft-*

ware Engineering, SE-17(3):241–258, March 1991.

[JL89] Jaffe, M.S. and Leveson, N.G. Implications of the man-machine interface for software requirements completeness in real-time, safety-critical software systems. *Proceedings of IFAC/IFIP SAFECOMP 89*, Dec. 1989.

[Lev95] Leveson, N.G. Safeware: System Safety and Computers. Addison-Wesley Publishing Co., 1995.

[Lev97] Leveson, N.G. Intent Specifications. in preparation.

[LHHR94] Leveson, N. G., M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, September 1994.

[LP97] Leveson, N.G. and Palmer, E. Identifying Indirect Mode Transitions: 'Oops, it didn't arm' as a case study. in preparation.

[MMR92] Madsen, M., Murphy, J.S., Rosso-Llopart, M. MAPS Software Requirements Specification. School of Computer Science, Carnegia Mellon University, June 1992.

[MLRPS97] Modugno, F., N. Leveson, J. Reese, K. Partridge, and S. Sandys. Integrated safety analysis of requirements specifications. *Third IEEE Interational Symposium on Requirements Engineering*, 1997.

[Pal96] Palmer, E. "oops, it didn't arm" – a case study of two automation surprises. NASA Technical Report, 1996.

[Per84] Perrow, C. *Normal Accidents: Living with High-Risk Technology.* Basic Books, Inc., New York, 1984.

[Ras90] Rasmussen, J. Human error and the problem of causality in analysis of accidents. In D.E. Broadbent, J. Reason, and A. Baddeley, editors, *Human Factors in Hazardous Situations*, pages 1–12, Clarendon Press, Oxford, 1990.

[SW95] Sarter, N. D. and D. Woods "How in the world did I ever get into that mode?": Mode error and awareness in supervisory control. *Human Factors 37*, 5–19.

[SW95] Sarter, N. D. and D. Woods Strong, silent, and out-of-the-loop. CSEL Report 95-TR-01, Ohio State University, February 1995.

[SW95] Sarter, N. D., Woods, D.D. and Billings, C.E. Automation Surprises. in G. Salvendy (Ed.) *Handbook of Human Factors/Ergonomics*, 2nd Edition, Wiley, New York, in press.