Investigating the Readability of Formal Specification Languages

by

Marc Kenton Zimmerman

B.S. Computer Engineering
University of California, Davis, 1997

M.S. Computer Science and Engineering
University of Washington, 1999

Submitted to the Department of Aeronautics and Astronautics in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 2001

Signature of Author: _____
Department of Aeronautics and Astronautics
May 11, 2001

Certified by: _____
Nancy G. Leveson
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: _____
Wallace E. VanderVelde
Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

Investigating the Readability of Formal Specification Languages

by

Marc Kenton Zimmerman

Submitted to the Department of Aeronautics and Astronautics
on May 11, 2001, in partial fulfillment
of the requirements for the degree of
Master of Science in Aeronautics and Astronautics

ABSTRACT

The readability of formal specification languages has been hypothesized as a limiting factor in the acceptance of formal methods by the industrial community. An experimental study was conducted to determine how various factors of state-based specification language design affect readability. Six factors were tested in all, including state machine representation, the expression of triggering conditions, macros, the use of internal events, hierarchies, and specification perspective. Subjects included computer scientists as well as aeronautical engineers in an effort to determine whether a correlation exists between a subject's background and his/her notational preferences. Such a correlation was difficult to establish based on our results, however.

This thesis contains a survey of various state-based specification languages in use today, and describes the design of a preliminary experimental study that was conducted to investigate the readability of various language design features, as well as the objective and subjective results obtained from the study. These results will serve as a starting point for more thorough experimentation in specification language readability.

Thesis Supervisor: Nancy G. Leveson
Title: Professor of Aeronautics and Astronautics

Acknowledgements

I would foremost like to thank my advisor, Prof. Nancy Leveson for not only her assistance with this thesis, but for her academic and personal guidance over the last few years: without your support I never would have finished.

Thanks is also due to my labmates and the rest of my subjects for enduring the experiment, and to Kristina for enduring the hours of complaining that accompanied its development. And thanks to Mirna for her truly exceptional data.

Thanks to my officemate Natasha for helping me with almost everything I've done at MIT, including this acknowledgements section. Remember that the leader of the world is no longer the country with the bravest soldiers but with the greatest scientists.

Of course, I have to thank my sister Kristine, who almost single-handedly kept my inbox full the past two years. Thanks for keeping me grounded.

I would like to thank Catherine, who taught me that there is more to movies than Total Recall. Your love and support over the years helped make this thesis possible. And a special thanks to my roommate Greg who helped me pretest this experiment, and showed me that even medical students can understand airplanes.

Finally I would like to thank my entire family for their love and support. And for living in California.

Table of Contents

Acknowledgements 3

List of Figures 6

List of Tables 7

Chapter 1: Introduction 8

Chapter 2: Background 12
- 2.1 State Machines 12
- 2.2 Related Work 15
  - 2.2.1 State-based Specification Languages 16
  - 2.2.2 Visualizations 33
  - 2.2.3 Human Experimentation 37

Chapter 3: Experiment Design 45
- 3.1 Features Selection 45
  - 3.1.1 Specifying the State Machine 46
  - 3.1.2 Specifying Events 47
  - 3.1.3 Specifying Transitions 48
- 3.2 Experiment Overview 50
- 3.3 Experiment Material 51
  - 3.3.1 Representation 53
  - 3.3.2 Conditions 56
  - 3.3.3 Macros 58
  - 3.3.4 Internal Events 60
  - 3.3.5 Hierarchies 62
  - 3.3.6 Perspective 62
- 3.4 Question Design 63

Chapter 4: Results / Discussion 66
- 4.1 Representation 68
- 4.2 Conditions 69
- 4.3 Macros 72
- 4.4 Internal Events 74
- 4.5 Hierarchies 78
- 4.6 Perspective 80
- 4.7 Experimental Error 82
- 4.8 Recommendations 83

Chapter 5: Conclusion 85
Bibliography 90
Appendix A: Experiment Questions 93
Appendix B: Representations Experiment Specifications 113
Appendix C: Conditions Experiment Specifications 136
Appendix D: Macros Experiment Specifications 145
Appendix E: Events Experiment Specifications 154
Appendix F: Hierarchies Experiment Specifications 162
Appendix G: Perspectives Experiment Specifications 164

List of Figures

Figure 1. State machine model for traffic light system..... 12
Figure 2. Traffic Light state machine operating in parallel with a Display state machine..... 13
Figure 3. State machine model using a superstate..... 14
Figure 4. Tabular representation of Traffic Light state machine..... 15
Figure 5. Overview of the Four Variable Model..... 16
Figure 6. SCR model for an altitude deviation detection system..... 17
Figure 7. Sample SCR specification for an altitude switch system..... 18
Figure 8. Statecharts model with History..... 19
Figure 9. Statecharts model of VCR system..... 20
Figure 10. AND-state notations in Statecharts..... 21
Figure 11. Overlapping state machines in Statecharts..... 21
Figure 12. Sample AND/OR table..... 24
Figure 13. Overview of SpecTRM-RL model..... 26
Figure 14. Sample SpecTRM-RL output definition..... 29
Figure 15. Operational Procedure Model..... 30
Figure 16. Sample Op-Proc table..... 32
Figure 17. *1 + 1 = 3* Effect of poor visualizations..... 36
Figure 18. Structural model to define object measurement..... 41
Figure 19. Sample textual specification..... 54
Figure 20. Sample specification table..... 54
Figure 21. Sample graphical specification..... 55
Figure 22. Sample graphical specification for trigger conditions..... 57
Figure 23. Sample logical specification for trigger conditions..... 57
Figure 24. Sample tabular specification for trigger conditions..... 58
Figure 25. Sample reference altitude specification with/without macros..... 59
Figure 26. Sample eventless specification..... 60
Figure 27. Sample specification with internal events..... 61
Figure 28. Histogram of Subject Performance for Representation Experiment... 68
Figure 29. Histogram of Subject Performance for Condition Experiment..... 70
Figure 30. Histogram of Subject Performance for Macros Experiment..... 72
Figure 31. Histogram of Subject Performance for Events Experiment..... 75
Figure 32. Responses to Question 13 of Events Experiment..... 76
Figure 33. Histogram of Subject Performance for Hierarchy Experiment..... 78
Figure 34. Transition from dead to time in hierarchical specification..... 79
Figure 35. Histogram of Subject Performance for Perspective Experiment..... 80

List of Tables

Table 1. Subjective Rankings for Condition Experiment..... 71
Table 2. Subjective Rankings for Macros Experiment..... 73
Table 3. Subjective Rankings for Events Experiment..... 77

1. Introduction

Software systems are relied on heavily in the aerospace field, and this reliance will only increase as their presence becomes more pervasive. However, this increased presence of software systems has been accompanied by a dramatic increase in system complexity. Aerospace systems being designed today are growing in scale and functionality, which is increasing the likelihood of subtle errors [19]. In light of this threat, we would like to provide some level of assurance for the system prior to its deployment. Ideally, we would like to do so early in the design process, as errors made during these stages are not only the most difficult to find, but can be the most expensive to correct. However, assuring the robustness of system specifications and designs is difficult to do using traditional software engineering practices.

One problem with testing large specifications is the typically large state-space involved. Formal methods and mathematical analysis theoretically present a way out of the dilemma posed by our inability to test even a small part of the enormous state space involved in most digital systems. They have the potential for both increasing safety and decreasing the cost of certifying flight-critical systems. The past 30 years have advanced the state of knowledge about formal methods to the point where many important problems can be solved. While formal methods are being applied to hardware in industry, the results of formal methods research for software has only rarely reached beyond the research lab and been used in industrial practice for day-to-day software development [35].

Formal methods are mathematically-based languages, techniques, and tools which typically accomplish one of two tasks in system development, specification or verification [5]. Formal specification involves creating a mathematical model of the system's behavior. This model usually contains behavioral information, but can include other types of information such as timing constraints and performance characteristics. Formal verification involves mathematically analyzing the model for various properties such as completeness (there is a behavior defined for every environment) and consistency (there is no non-deterministic behavior specified). This work will focus specifically on

issues related to formal specification, in an effort to maximize the potential benefits of this approach to specifying system behavior. While a formal method is usually better suited for a particular type of system, several different types of formal methods have been established in recent years, allowing us to specify and analyze a diverse group of systems [2].

As mentioned before, formal specifications afford us several benefits. First, they provide a precise, unambiguous specification of the system's behavior. System specifications are typically written using English prose, which is of course easy to use and conducive to creating specifications quickly. However, it can be ambiguous and wordy, which are obviously problematic qualities when working with others to develop a specification. Formal specification languages possess a defined syntax and semantics which alleviate this confusion and ambiguity. Along the same lines, by providing a single, explicit language of communication, a formal method can help bridge gaps between a diverse set of system designers and developers.

Formal methods can also help decrease cost. Errors made during the specification stage are not only the most difficult to find but can be the most expensive to correct. Specification errors are typically not discovered until systems are designed or even implemented. At this point, designers must trace back to the specification, fix the error, and redesign the system. This process can not only be a tedious one, but an expensive one as well. Because formal methods operate directly on a system specification, we can prevent these added costs. In addition, because formal specifications are defined mathematically, they are conducive to automated analysis. Properties such as completeness and consistency can then be assured quickly with an automated tool, eliminating the potential for human error.

Despite these benefits, formal methods have found a tremendous lack of interest in industry. Several reasons may be hypothesized for this lack of wide-spread adoption. First, most formal languages are based on discrete math and/or logic. However, engineers are typically not trained in discrete mathematic. Further, the notations used in these languages are often not as concise or as parsimonious as their continuous math counterparts. So while a control law can be represented as a differential equation, the

discrete mode logic for a flight management system might require hundreds of pages of formal logic to specify. The review of such specifications by domain experts is a daunting task. In addition, the tools provided for formal analysis of such specifications are often difficult for engineers to use: They may require in-depth knowledge of discrete mathematics, and may require domain experts to translate the problem as they understand it in their domain of expertise into another domain; for example, they may be required to specify a set of axioms that describe the domain and to restate the problem in terms of theorems and lemmas that they must then prove, perhaps with some assistance from an automated tool [35]. The scope and scalability of formal methods remain additional concerns both in industrial and academic communities. As mentioned before, there has been success with applying formal methods to hardware systems analysis. Hardware systems typically exhibit a high degree of regularity, where you may find the same component duplicated thousands of times on the same chip. Software systems, on the other hand, very rarely exhibit this kind of regularity, and so there has as yet been only limited success with applying formal methods to complex systems.

However, one of the biggest drawbacks to using formal languages is that they simply are not readable. Readability is arguably one of the most important properties of any specification [20]. The specification must not only be reviewable by those trained in the language, but must be readable by a large variety of people with diverse backgrounds and expertise including system designers and developers, customers, users, certifiers, etc. Readability by a general audience allows all involved parties to discuss and analyze a specification using a single common model. Furthermore, our experience in analyzing formal specifications for complex systems suggests that the most significant errors and omissions will be found by human experts rather than automated tools [35]. This observation does not mean that automated tools are not useful and important in finding some types of errors, especially those that require tedious checks, but humans are required to determine whether a specification conforms with engineering expectations and requirements. In addition, even those design or specifications flaws found by tools will need to be evaluated by human experts. Therefore, readability of system specifications is a requirement not only for human understanding of complex models but also for human processing of the analysis results.

Readability is also desirable as it has associated with it a short familiarization time. Certainly any specification language is going to require some training in order to understand and use. However, particularly with respect to review, this time cannot be too long before it becomes impractical to coordinate the considerable amount of reviewing that leads to high-quality specifications and software. It can require as much as 3-6 months training before an engineer can use some formal languages effectively. This amount of training not only makes it difficult to review formal specifications, but also discourages the addition of engineers to the specification and design teams.

In an effort to increase the practicality of formal methods, this thesis deals specifically with the problem of specification language readability. Designing a specification language that is readable by a general audience is a difficult problem, and one that has received little attention in research communities. To date, there has been little empirical evidence produced to guide specification language design that supports readability. We have designed and completed an experiment which has provided preliminary results to help us understand which notations and conventions are more conducive to readability. We specifically included subjects with both computer science and engineering backgrounds, as these are two fields that can directly benefit from the use of formal methods. Understanding which notations are more readable, and to which audiences, can help us create specification languages that are more expressive and effective, allowing formal methods to become a more attractive alternative for the industrial community.

This section has hopefully provided an introduction to the use of formal methods, its benefits and drawbacks, as well as motivation for this thesis. The next chapter will provide a detailed overview of a specific class of formal languages studied in this work – state-based specification languages – as well as a survey of related work in language readability. We will then discuss the design and implementation of our experiment, followed by an analysis of our results. The final chapter will summarize the conclusions drawn from this work, as well as identify some directions for future work in the area.

2. Background

Formal specifications involve the creation of an underlying mathematical model. This model can take on several different forms. In this work, we will focus specifically on specification languages that use an underlying state machine model.

2.1 State Machines

A state machine models system behavior in terms of states and transitions between them. Figure 1 shows a state machine model for a simple traffic light system, called *Traffic Light*. *Traffic Light* has three different states – Green, Yellow, and Red – which are denoted by circles. The arrows between them denote transitions from one state to another. Transitions have a source and destination state, and are labeled with triggers and possibly outputs as well. The trigger for a transition is made up of events and/or conditions. An *event* is an occurrence in time, and a *condition* is a statement that can have the value True or False.

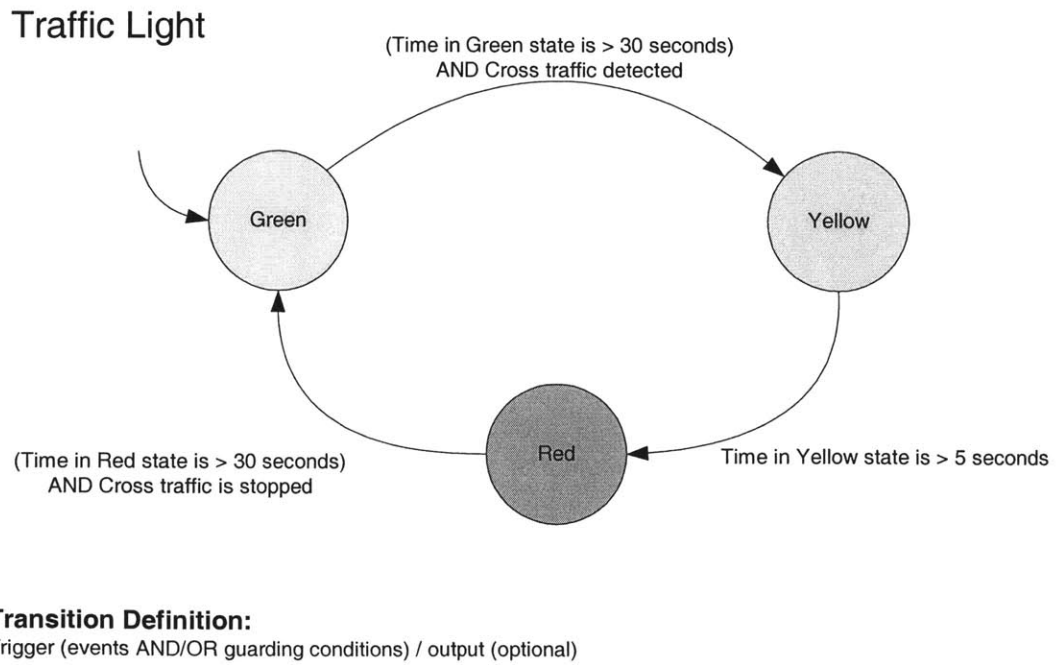


Figure 1. State machine model for traffic light system.

A transition will be taken from its source state to its destination state when the corresponding trigger occurs. For example, the system in figure 1 will transition from Green to Yellow when a "Time in Green State is > 30 seconds" event occurs and the condition (i.e. cross traffic has been detected) is true. An arrow with no source state denotes the starting state of the system (Green, in figure 1).

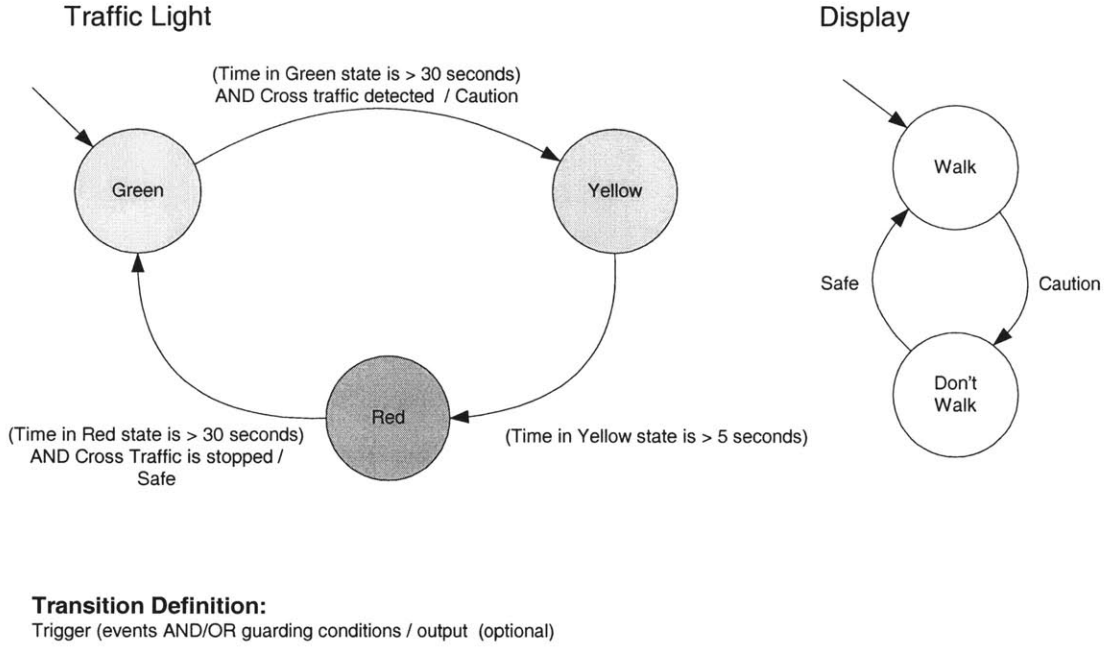


Figure 2. Traffic light state machine operating in parallel with a display state machine.

As mentioned before, events can serve as triggers for transitions. However, events can also be used as outputs. A transition can produce an output event, which in turn can trigger another transition in the system. For example, figure 2 shows a state machine model for a Walk/Don't Walk display that operates in parallel with the traffic light system described earlier. When the traffic light state machine transitions from state Green to Yellow, there is now an output event "Caution" generated. This event in turn will trigger a transition from Walk to Don't Walk in the Display state machine.

State machines can also contain a hierarchy in the form of *superstates*. For example, the states and transitions of the traffic light / display state machines can be grouped together to form the superstate "On," as shown in figure 3. When we are in superstate "On," the traffic light state machine behaves just as described earlier.

However, if the event "Power-Off" occurs at any time, the state machine in figure 3 will transition to Off. So, in essence we have a two-level hierarchy in this example. At the highest level, we have the On-Off state machine. However, within the On state, we find another modular state machine – the traffic light – which is active only when the system is in the On state. There are infinitely many possible levels to a state machine hierarchy, which allows us to model complex systems.

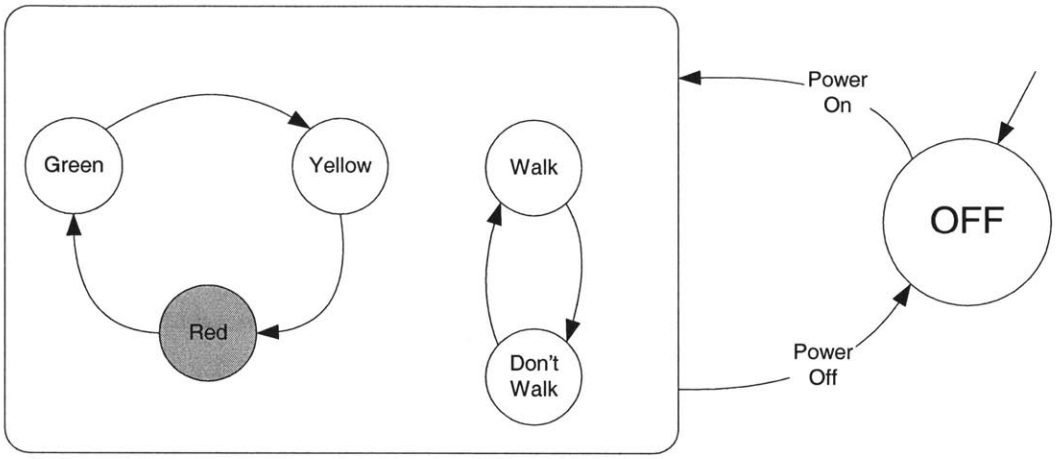


Figure 3. State machine model using a superstate.

It is important to note that state machines are an abstract model that can be described in several ways. Figure 1 describes the traffic light state machine graphically. It could also be described textually, or in a table format, as in figure 4. Similarly, while the trigger conditions in these examples are simple, complex conditions can be expressed in a variety of ways (e.g. using propositional logic, graphically, in a table, etc.). There are several ways to describe the parts of a state machine, each offering different potential benefits. What differentiates one state-based specification language from another is simply the way that it chooses to represent the parts of this underlying model. This overview of state machines should provide adequate background for this thesis. For more information regarding the formal mathematical definition of the state machine model, please refer to [29].

Current State	Green	Yellow	Red
New State	Yellow	Red	Green
Trigger	(Time in Green > 30 seconds) AND Cross Traffic is detected	Time in Yellow > 5 seconds	(Time in Red > 30 seconds) AND Cross Traffic is Stopped
Output Event	NA	NA	NA

Figure 4. Tabular representation of Traffic Light state machine.

As mentioned before, this work deals specifically with state machine-based specification languages, primarily because they seem to be more likely to be adopted by industry than other formal methods. The state machine model is well researched, and supported by several proven analysis techniques. It also possesses several features that make it a readable and understandable description of system behavior, arguably more than other formal models. First, state machines do not require knowledge of their formal foundation in order to be used effectively, in contrast to other formal models. In fact, one requires little knowledge more than that presented in the traffic light example discussed above. In addition, we hypothesize that state-machine models are a natural way for engineers to think about control systems. The behaviors of many systems are easily described using modes (states) and transitions between them. This type of description can readily be expressed by a state machine.

2.2 Related Work

Program readability has been researched for years, as it is a useful property to have when debugging software. However, from a system design and evolution perspective, specification readability is paramount. The aim of this work is to determine those features of specification language design, those representations of the parts of a state machine, that can increase the readability and comprehensibility of system specifications for an appropriate audience. We attempt to do so by human experimentation, which seems to be the most appropriate and objective means of assessing readability.

Though there is no work known to date that deals with the specific topic of specification language readability, the ideas developed in this thesis are drawn from several related areas of research including specification language development, visualization design, and computer science experimentation. Each of these related areas of work are supported by several research efforts, which will be described in the following sections.

2.2.1 State-based Specification Languages

Employing formal methods during the specification stage is arguably one of the most effective ways of reducing specification errors. Several state-based specification languages have been developed to date for this particular purpose. An overview of the most significant contributions to this effort will be discussed in this section. Each language presented employs the same underlying state machine model – they are differentiated by the notations used to express the model to the reader.

SCR

Software Cost Reduction (SCR) is a state based specification language and process originally designed to be used by engineers. It was actually introduced roughly 20 years ago, and has been undergoing refinement since, based on experiences with applying the method to actual system development. The SCR method has proved effective upon application to several avionics systems, such as the A-7 Operational Flight Program and Lockheed's C-130J Operational Flight Program (OFP). Some of these systems comprised over 100,000 lines of code, which provides partial evidence of the method's scalability [15].

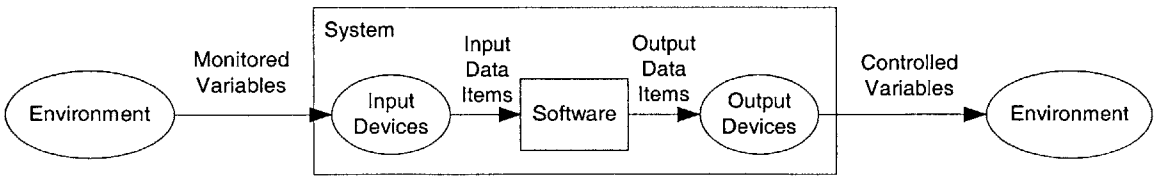


Figure 5. Overview of the Four Variable Model.

The SCR method is based on a Four Variable Model, which is simply basic control theory. The model is shown in figure 5. According to this model, system behavior is described using mathematical relations over four sets of variables – monitored variables, controlled variables, input data, and output data. Monitored variables correspond to elements of the environment that can effect the system (or in particular, the input devices). Controlled variables are those parts of the environment that the system controls (via output devices). Input and output data describe the inputs and outputs to the control software of our system.

To specify these relationships in the system, SCR uses four different constructs – mode classes, terms, conditions, and events. A mode class is simply a state machine, whose transitions are triggered by events, and whose states are determined by the values of monitored variables. A term is "an auxiliary function defined on input variables, modes, or other terms that helps make the specification concise. A condition is a predicate defined on one or more system entities at some point in time. An event occurs when any system entity changes value," and so individual events are not explicitly defined as they are in other languages [15, pp.5-6]. The following example serves to illustrate these constructs.

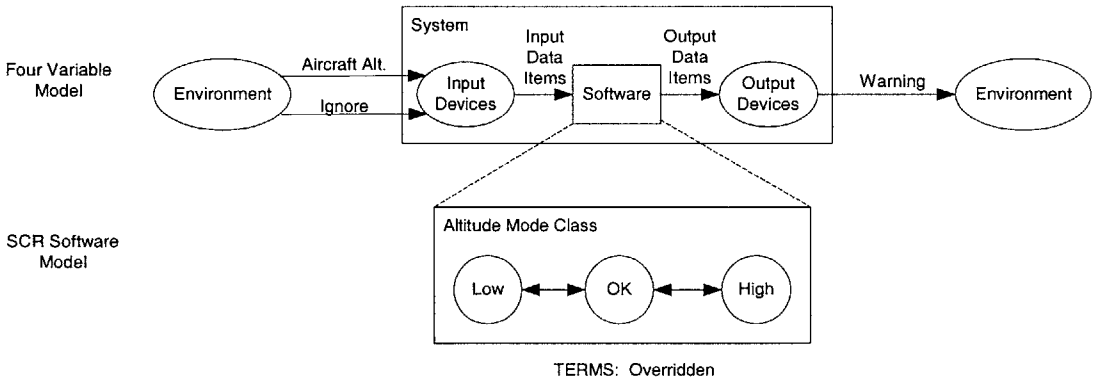


Figure 6. SCR model for altitude deviation detection system.

Figure 6 shows a model for a simple altitude deviation system. Depending on the state of the system and the environment, the system will determine that the aircraft's altitude is too high, permitted, or too low, and if necessary, it will issue a command to adjust the altitude. Altitude is a mode class, or state machine, composed of three states

(Too high, Permitted, Too low). Altitude will transition between these states based on the value of the aircraft's vertical position, which is a monitored variable of the system. There is a term, Overridden, which is a function conditioned on the state of the mode class and other inputs, and will determine whether any altitude warning should be overridden (for example, if the pilot intentionally commands the aircraft Too High). Warning is a controlled variable for the system, which will be activated when it is desirable to send a warning signal to the environment (e.g. the cockpit).

Old Mode	Event	New Mode
Low	@T(Aircraft Alt. > 1000ft.)	OK
OK	@T(Aircraft Alt. ≤ 1000ft.)	Low
OK	@T(Aircraft Alt. ≥ 40,000ft.)	High
High	@T(Aircraft Alt. < 40,000ft.)	OK

Altitude Mode Class (state machine)

Mode	Events	
High	False	@T(In Mode)
Low ∨ OK	@T(Ignore = On)	@T(Ignore = Off)
Overridden	**True**	**False**

Overridden Term

Mode	Conditions	
High	False	True
Low	Overridden = True	Overridden = False
OK	True	False
Warning	**Off**	**On**

Warning Controlled Variable

Figure 7. Sample SCR specification for an altitude switch system.

SCR specifications use a tabular notation to specify the behavior of the system in terms of these constructs. Figure 7 shows a tabular description of the Altitude mode class, the Overridden term, and the Warning controlled variable. The first table specifies the behavior of the Altitude mode class as a function of its current state and input events (which are generated whenever the aircraft altitude input changes value). For example, If the mode class is in state *OK* and the altitude becomes greater than 40,000 ft, the mode class will transition to state *High*. The second table also uses events to specify the

Overridden term as a function of the mode class Altitude and the input Ignore. According to the table, the warning can be overridden if the mode class is in state *Low* and the system receives an *Ignore* = *True* signal from the pilot. The final table specifies the behavior of the Warning variable, as a function of the mode class Altitude and the term Overridden. For example, the warning will be *On* if the mode class is in state *Low* and Overridden is *False*.

This overview should provide adequate exposure to the SCR method in so far as readability is concerned. For a more detailed description of the underlying mathematical model, the reader is referred to [15].

Statecharts

Statecharts is a visual formalism used to specify complex reactive systems [11]. Based on a more general model, the higraph, the statecharts formalism is actually very similar to the state machine notation introduced earlier. States are represented using boxes, or rounded rectangles, while transitions between them are described using arrows. The arrows are labeled with triggers for the transitions they describe, which must include an event, and may include a condition. Statecharts also makes use of a superstate notation to describe hierarchies. A superstate is denoted by drawing a box around the states of which it is composed. This notation allows a level of abstraction which is useful in describing system behavior.

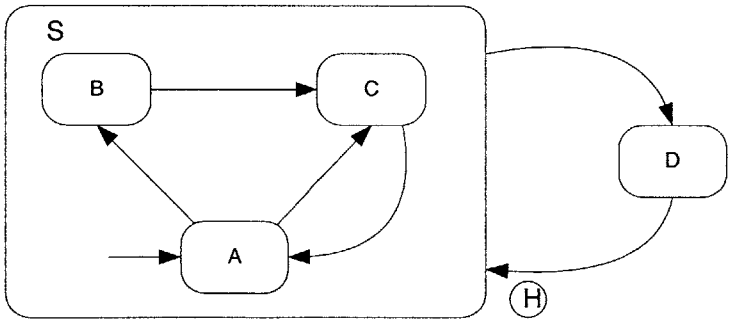


Figure 8. Statecharts model with History.

The statecharts formalism is unique in that incorporates memory into its specifications by means of the history notation. If a superstate is exited and then re-

entered, history allows a state machine to re-enter the superstate at the most recently visited state within the superstate, rather than the superstate's default state. For example, there is a history transition into superstate S of figure 8. If superstate S is in state *C* when it is exited, then it will return to *C* if it is re-entered. The first time S is entered, it will be in the default state, *A*. While the history notation implies a sense of memory, it is important to realize that a functionally equivalent (but much more complex) statecharts specification can be produced without history.

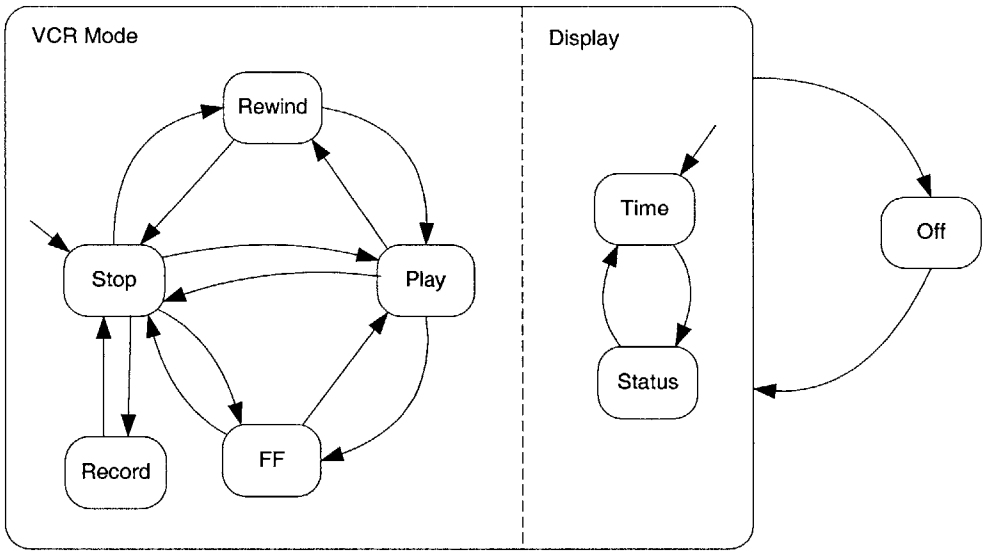


Figure 9. Statecharts model of VCR System.

Often times, systems are composed of several subsystems, each of which can be represented by a state machine. It should be clear that the state space (i.e. the number of states) in the composite system can blow up very quickly with the addition of subsystems, as the total state space is actually the cross product of the state spaces of each component state machine [12]. However, we can describe the total space by describing each component state machine. In this way, the total state space is not generated, but it is clear how it *could* be generated if necessary. Statecharts is often credited with creating the idea of the AND state, whereby two or more state machines operate in parallel. An AND state is described by drawing a box around all component state machines, and then separating each component state by a dashed line. Figure 9 shows an AND state *On* composed of the VCR Mode and Display state machines. When

the system transitions to *On*, both component state machines are entered as well. AND state composition has proven to be a necessity when modeling any realistically complex system, and has been incorporated into virtually all state based specification languages developed since.

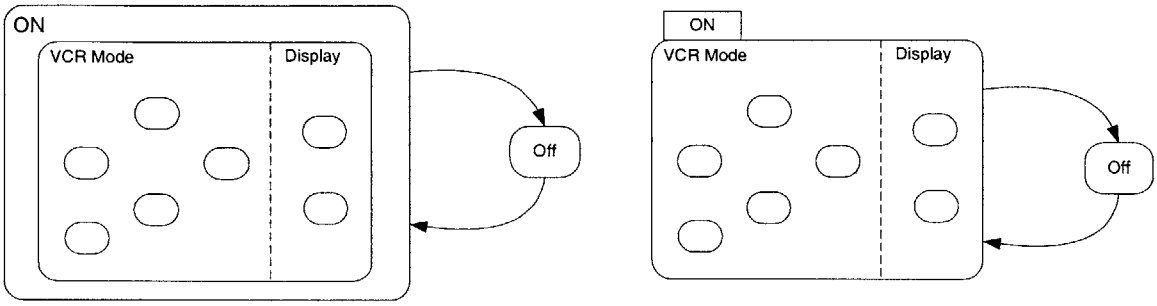


Figure 10. AND-state notations in Statecharts.

The statecharts formalism is not clear on how AND states are to be labeled, however [11]. One possibility is to draw a new box around the entire AND state, and include the label in the new box, another is to include a tag on the AND state with the label. An example of both is shown in figure 10.

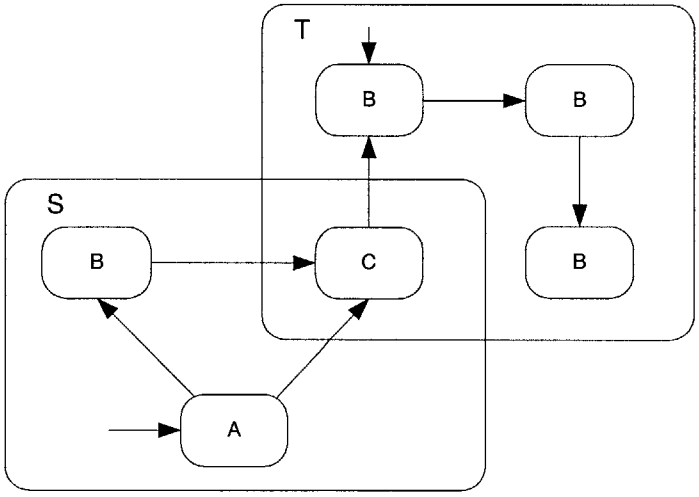


Figure 11. Overlapping state machines in Statecharts.

Statecharts makes use of internally broadcast events to allow communication within and between state machines in a system. As described before, these events are produced as outputs of transitions. Statecharts describes internally broadcast events on transition arrows alongside triggering conditions. When an event is generated as an

output of a transition, it is broadcast throughout the specification, and can then trigger other transitions in the model.

While this definition of statecharts allows the specification of some fairly complex mechanisms, there is also the possibility of extending the formalism to express more complex behaviors. For example, one possible future extension involves overlapping superstates, as shown in figure 11. This notation can potentially capture conceptual similarities among superstates, or may simply be used to economize the expression of certain state exits. Another possible extension to the formalism is to incorporate temporal logic, which is often used in the specification of concurrent systems [30]. This approach can take on a couple of different forms. First, based on given constraints specified in temporal logic, we can develop an equivalent statecharts specification, and then verify that the statecharts specification does in fact adhere to the specified constraints. One possible solution to this problem may involve extending previously derived methods to verify state machines against temporal logic constraints. Another way to incorporate temporal logic into statecharts development is to possibly derive a statecharts specification from a temporal logic specification. Such an ability would make the use of statecharts more attractive to those that are more comfortable with creating temporal logic specifications. For a more detailed discussion of statecharts development, as well as the defined semantics of the language, please refer to [11].

RSML

In the early 1990's, researchers from the Safety-Critical Systems Research Group at the University of California, Irvine set out to formally specify the requirements for TCAS (Traffic Collision Avoidance System) II. The system was specified using RSML (Requirements State Machine Language), a state machine based specification language based on Statecharts [14]. RSML incorporates several features of the Statecharts formalism, but exhibits new ones as well, to account for previous difficulties with applying Statecharts to reactive systems. The TCAS experience was significant in that it not only showed that RSML can be used to specify complex systems, but also that formal specification languages can be readable by an engineering audience with limited training

[18]. In fact, the RSML specification was eventually adopted as the official specification for TCAS II.

Like Statecharts, RSML permits states of a state machine to be grouped together into superstates. As explained before, this helps reduce the number of transitions that are explicitly defined. RSML also makes use of AND-decomposition (or the orthogonal product, in Statecharts), where state machines can operate in parallel. This allows a much more concise representation of a system's total state space.

Another significant feature of RSML borrowed from Statecharts is the use of internally broadcast events [13]. Transitions in RSML consist of a source and destination state, a location (of the transition within the state machine), a triggering event, guarding condition (if necessary), and an output action. A transition is taken when a triggering event occurs and the corresponding guarding condition is true at the same time. After a transition is triggered in a state machine, an internal event may be generated as an output action. Other state machines are notified of this event via the broadcast mechanism. Unlike Statecharts, RSML was designed to specify a system's blackbox behavior – RSML specifications describe a system solely in terms of inputs and outputs, and the relation between them. Furthermore, the relation between inputs and outputs is given only in terms of variables and conditions that are externally visible. Internal events are therefore used much less liberally than in Statecharts. In fact, they are used only to order the triggering of transitions in a state machine, a function that can be likened to parentheses/brackets in the evaluation of mathematical expressions.

By specifying purely blackbox behavior, RSML can potentially increase the readability of its specifications. Leveson et al. hypothesize that readability and reviewability are enhanced by minimizing semantic distance between the reviewer's mental model of the system and the system specification model [18]. Semantic distance can loosely be defined as the amount of effort required to translate from one model to another. Leveson et al. believe that the application expert's ability to find errors in a requirements specification can be enhanced by reducing the semantic distance between their understanding of the required process control behavior and the specification of that behavior. This, in turn, implies that specifications should use familiar engineering

notations and that they be written in terms of the externally observable behavior of the component being specified. Any information related only to the implementation of that behavior should not be included. That is, the specification should be blackbox. Thus RSML specifications include only the input/output function being computed by the component (the transfer function, in engineering terminology) and do not include any information about the internal design of the component or how that externally visible behavior is actually realized. In fact, the blackbox behavior might be achieved through the use of hardware or software.

The notation used to express guarding conditions serves as another distinction between RSML and Statecharts. Where Statecharts uses simple predicate calculus to represent guarding conditions, RSML makes use of a mechanism known as an AND/OR table, an example of which is shown in figure 12.

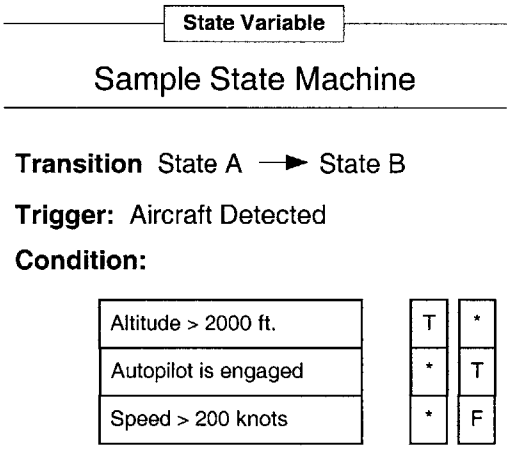


Figure 12. Sample AND/OR table.

AND/OR tables are concise representations of propositional logic [13]. The first column is a set of boolean phrases – each of which evaluate to True or False – and the additional columns represent various conjunctions of those phrases. If all of the elements of a column evaluate to True, then the column evaluates to True. If *any* of the columns evaluates to True, then the entire table evaluates to True, and a transition is triggered. For example, in figure 12 we see that the state variable Sample State Machine will transition from *State A* to *State B* if the *Aircraft Detected* event occurs and the AND/OR table evaluates to True. This will happen if "Altitude > 2000ft" is True, OR if "Autopilot

is engaged" is True and "Speed > 200 knots" is False. A `*' denotes a "Don't Care" (i.e. the input can have any value). This representation of guarding conditions has been found to be readable by engineers with a limited amount of training.

RSML also allows for simplification of guarding conditions with macros. Macros in a state-based specification language function basically as they do in any programming language. They are simply pieces of logic (an AND/OR table, in RSML) that, once specified, can be referred to by name in other pieces of logic (AND/OR tables). This modularization is conducive to specifying simpler blocks of logic which are theoretically easier to read and understand. Using appropriate naming conventions when specifying macros can also lead to more readable specifications. Previous work with formal specifications has prompted researchers to conclude that macros are a necessity if formal languages are ever to scale to realistic systems [35].

For a detailed discussion RSML's functional framework, the reader is referred to [13].

SpecTRM-RL

Since creating the TCAS specification, Leveson et al. have created specifications of real systems, experimenting with specification language features with respect to usability. SpecTRM-RL (Specification Tools and Requirements Methodology—Requirements Language) is the latest manifestation of the lessons they have learned to date [20].

SpecTRM-RL is a state-based specification language that shares many similarities with its predecessor language, RSML, that contribute to its readability including superstates, AND decomposition, and the use of AND/OR tables to express guarding conditions. Also, like RSML, SpecTRM-RL was designed to specify process-control systems, and in particular the blackbox behavior of these systems. However, where RSML simply allows blackbox specifications, SpecTRM-RL actually enforces them [21]. Previous experience with RSML (and other general modeling languages) has shown that it is difficult to exclude design elements from (blackbox) system specifications. Therefore, SpecTRM-RL was not designed as a general modeling language, but as a blackbox specification language (like SCR, above). Certain features that are not required

for blackbox behavior specifications are not included in the language, while other features that encourage blackbox specifications have been included. For example, SpecTRM-RL introduced a "mode" abstraction, which is a functional behavior that is externally visible, and therefore blackbox.

SpecTRM-RL exhibits several other distinctions from RSML that can potentially enhance the readability of the language. Most importantly, SpecTRM-RL does not use internally broadcast events. Internally broadcast events proved to be the cause of the majority of errors found during the review of the TCAS II specification [21]. Furthermore, internally broadcast events can be used to include design details and programming features like counters, which are inappropriate for blackbox specifications. Therefore, they were effectively removed from the language. Instead, SpecTRM-RL relies on data dependencies in the specification to determine the order of execution of the state machine model.

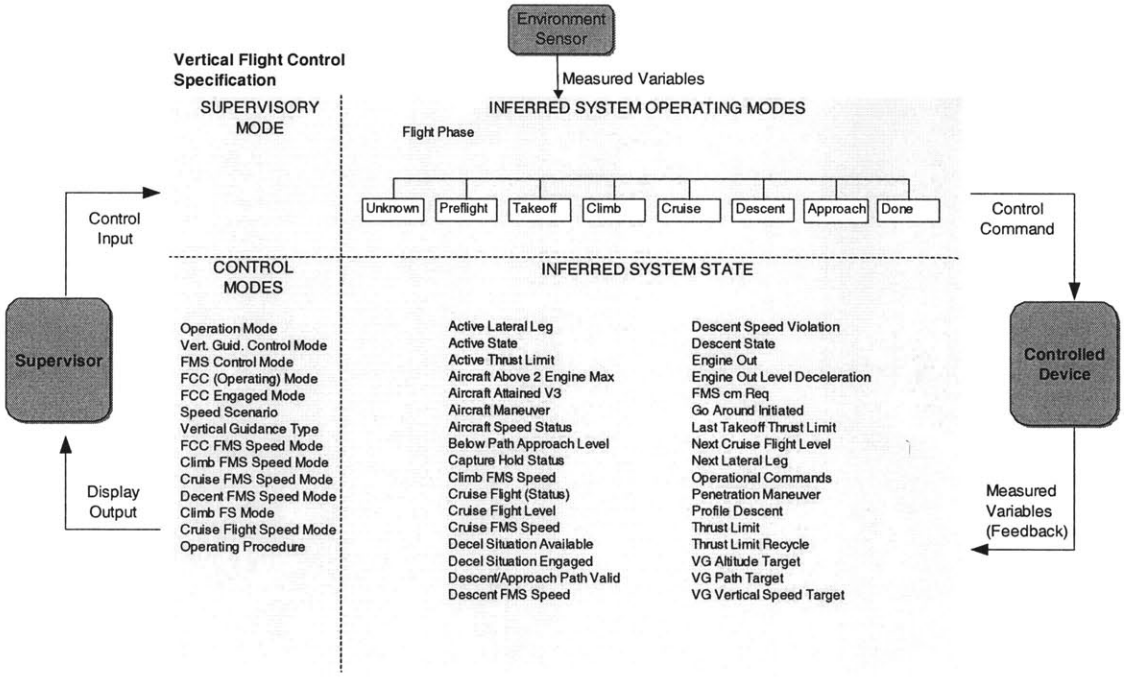


Figure 13. Overview of SpecTRM-RL Specification.

SpecTRM-RL also differentiates itself from RSML by limiting the semantic domain of the language, which can hypothetically increase the language's readability. SpecTRM-RL was designed primarily for process-control systems. The high-levels of

the specification look very similar to process-control diagrams. Figure 13 shows the basic format of the specification [35]. This example in particular applies to a vertical navigation system. There are four main parts: (1) a specification of the supervisory modes of the controller being modeled, (2) a specification of its control modes (3) a model of the controlled process (or *plant* in control theory terminology) that includes the inferred operating modes and system state (these are inferred from the measured inputs), and (4) a specification of the inputs and outputs to the controller. The graphical notation mimics the typical engineering drawing of a control loop.

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

The supervisory interface consists of a model of the operator controls and a model of the displays or other means of communication by which the component relays information to the supervisor. Note that the interface models are simply the logical view that the controller has of the interfaces – the real state of the interface may be inconsistent with the assumed state due to various types of design flaws or failures. By separating the assumed interface from the real interface, SpecTRM-RL is able to model and analyze the effects of various types of errors and failures (e.g., communication errors or display hardware failures). In addition, separating the physical design of the interface from the logical design (required content) facilitates changes and allow parallel development of the software and the interface design. During development, mockups of the physical GUI or interface design can be generated and tested using the output of the SpecTRM-RL simulator.

Supervisory modes are used in specifying information about the current supervisor of the controller and are useful when a component may have multiple supervisors at any time. For example, a flight control computer in an aircraft may get inputs from the flight management computer and also directly from the pilot. Required behavior may differ

depending on which supervisory mode is currently in effect. Mode-awareness errors related to confusion in coordination between multiple supervisors can be defined (and the potential for such errors theoretically identified from the models) in terms of these supervisory modes. In systems with complex displays (such as Air Traffic Control systems), it may also be useful to define various *display modes*.

The bottom left quadrant of figure 13 provides information about the *control modes* for the controller itself. These are not internal states of the controller (which are not included in our specifications) but simply represent externally visible behavior about the controller's modes of operation. *Control Modes* are used in describing the required behavior of the controller. Modern avionics systems may have dozens of modes. Control modes may be used in the interpretation of the component's interfaces or to describe the component's required process-control behavior.

The right half of the controller model represents inferred information about the operating modes and states of the controlled system (the *plant* in control theory terminology). A simple plant model may include only a few relevant state variables. If the controlled process or component is complex, the model of the controlled process may be represented in terms of its operational modes and the states of its subcomponents. *Operational modes* are useful in specifying sets of related behaviors of the controlled-system (plant) model. For example, it may be helpful to define the operational state of an aircraft in terms of it being in takeoff, climb, cruise, descent, or landing mode.

In a hierarchical control system, the controlled process may itself be a controller of another process. For example, the flight management system may be controlled by a pilot and may issue commands to a flight control computer, which issues commands to an engine controller. If, during the design process, components that already exist are used, then those plug-in component models can be inserted into the SpecTRM-RL process model. Additionally, parts of a SpecTRM-RL model can be reused or changed to represent different members of a product family.

Each piece of the high-level SpecTRM-RL model needs to be specified in detail. As an example, figure 14 shows the specification for the Target Altitude output. The conditions under which outputs are assigned values are described using the AND/OR

tables of RSML, with slight modifications. Each AND/OR table in SpecTRM-RL is divided into two parts, *Control Modes* and *State Values*. The Control Modes section describes the value of the control modes necessary for the transition, while the State Values section describes the values of and conditions on inputs and state variables. This distinction allows the reader to better understand each mode of the system's behavior, and has been found helpful in detecting specification errors – particularly omissions.

Output Variable

Target Altitude

Type: INTEGER
Destination: TBD
Initiation Delay: 0 milliseconds
Completion Deadline: TBD
Exception Handling: None specified
References: N/A

Feedback Information:
Variables: UNDEFINED
Values: UNDEFINED
Min time between outputs: 10 MHz
Max time between outputs: UNDEFINED
Exception Handling: None Specified

:= Vertical Guidance Climb Target Altitude **IF**

TRIGGERING CONDITION

Control Modes	Vertical Guidance Operating Procedure IN_STATE Airmass Ascent	T	T	*	*
	Vertical Guidance Operating Procedure IN_STATE Climb InterLev	*	*	T	T
State Values	Flight Phase IN_STATE Takeoff	T	*	T	*
	Flight Phase IN_STATE Climb	*	T	*	T

Figure 14. Sample SpecTRM-RL output definition.

To assist with cognitive manageability, SpecTRM-RL also requires that its specifications contain a graphical overview. An example of the information contained in the overview can be found in the *Flight Phase* description in figure 13. The overview simply contains a list of all component state machines in the specification as well as the possible states each state machine can be in. For example, the flight phase state machine can be in state Takeoff, Climb, Cruise, etc.

A final innovation of SpecTRM-RL is that it is well integrated into a complete system engineering methodology called intent specification which combines both formal and informal system requirements [23]. Intent specifications are a five-level approach to

system specification that traces development from the highest level, goal and requirements definition, down to implementation details. Levels of the specification abstract on intent – each level explains "why" for the level below. SpecTRM-RL models correspond to level three of the intent specification, specifying the system's blackbox behavior. Providing the reader with the ability to trace requirements in a SpecTRM-RL model up to higher levels, and thereby understand the rationale behind its development, can theoretically increase the readability of the language. For a more complete description of the language, the reader is referred to [22].

Op-Proc Tables

Like the other specification languages described in this section, the OpProc Table is a state-machine based notation developed to specify embedded reactive systems. OpProc Tables are actually used to describe the operational procedure model, illustrated in figure 15 [28].

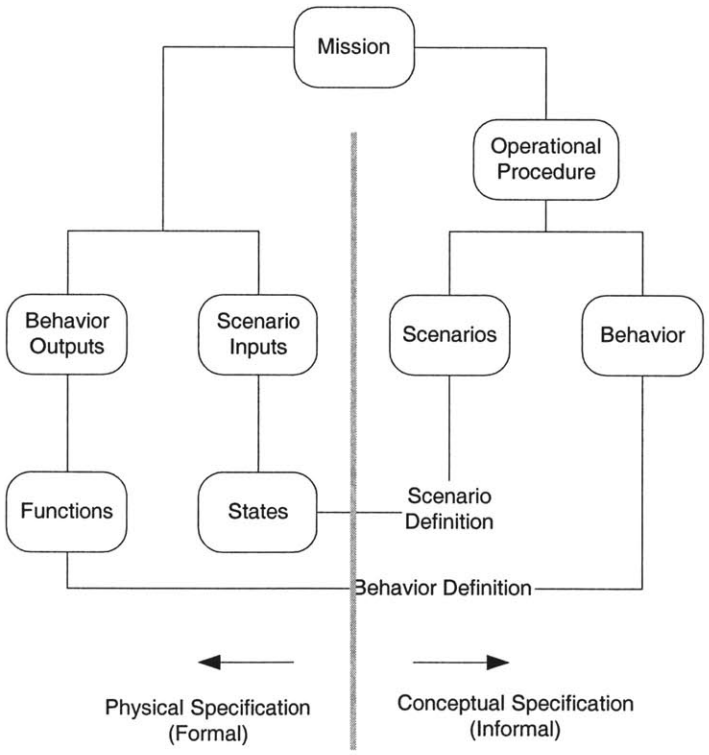


Figure 15. Operational Procedure Model.

There are two parts to the operational procedure model, the conceptual specification and the physical specification. The conceptual specification describes the system as a set of operational procedures which are called upon to successfully achieve the goals of the system. This part of the model is an informal specification, describing operational procedures (behaviors) and the conditions (scenarios) under which each procedure is called using English prose.

The physical specification contains roughly the same type of information, but expresses it formally as the relationship between inputs and outputs of the system. This relationship is described using an OpProc Table. The table defines scenarios based on states of the system inputs, and then based on the appropriate scenario, determines which operational procedure should be invoked. The behavior of each procedure is described by specifying the behavior of the system outputs under each procedure.

As seen in figure 15, the conceptual and physical specifications are linked together in two ways, by the scenario definition and behavior definition. Both specifications describe the same scenarios and behaviors, one formally, and the other informally. Requiring this type of redundancy can increase the readability of the specification.

Like SCR, the OpProc formalism uses a tabular format to describe the state machine underlying the physical specification. Figure 16 shows an example of an OpProc table. There are three parts to the table, the Operational Procedures, the Operational Scenarios, and the Operational Behaviors. The names of the Operational Procedures are found in the top row of the table, i.e. *Cruise Procedure*, *Climb Procedure* and *Descent Procedure*. The next section of the table defines Operational Scenarios according to the states of the system inputs. The first two columns describe the inputs to the system and the discrete values (or states) that each can take on. The following columns define scenarios as specific combinations of input values, and detail which scenarios will invoke a given operational procedure. For example, in figure 16, scenario 2 is defined to be the case where the altitude is less than the Cruise Flight Level, and the current operational procedure is Cruise. The Climb Operational Procedure will be

invoked if scenario 2 is true. Like RSML and SpecTRM-RL, OpProc tables enforce AND/OR decomposition of conditions.

OpProcs			Cruise Procedure	Climb Procedure	Descent Procedure
Scenarios	Inputs	States	Scenario 1	Scenario 2	Scenario 3
	Altitude	< Cruise Flight Level, Cruise Flight Level, > Cruise Flight Level	Cruise Flight Level	< Cruise Flight Level	> Cruise Flight Level
	Current OpProc	Cruise, Climb, Descent	Climb, Descent	Cruise	Cruise
Behaviors	Outputs	Functions	Behavior 1	Behavior 2	Behavior 3
	Output 1	f1, f2	f1	f2	f2
	Output 2	f1, f3	f1	f3	f3

Figure 16. Sample OpProc table.

The final section of the table defines the behavior of each operational procedure by specifying the behavior of each output under each procedure. Each output is associated with a given set of functions that can be used to determine its value. For example, in figure 16, output 2 can be defined by function f1 or f3. The specific combination of functions that generates each output defines the behavior of an operational procedure.

To assist with the task of developing a mental model, and hence to increase the notation's comprehensibility, OpProc tables supports the following three abstractions: conceptualization, (mathematical) simplification, and organization. Conceptualization refers to the inclusion of the conceptual specification, which should represent an easily read English language description of the system's behavior. Mathematical simplification helps to reduce the number of scenarios that need to be defined by allowing AND/OR decomposition of system inputs and by converting the underlying model from a fully-connected state machine to one that is partially connected. Allowing AND/OR decomposition of the inputs simply means that different values for a system input can be grouped together into the same scenario definition. For example, in figure 16, the Current OpProc input can be Climb or Descent in scenario 1. Organization, the final type

of abstraction, can take on several forms, including aggregation (placing all operational procedures in one cohesive table), and generalization (referring to different tables to define the behavior of an Operational Procedure). Generalization in essence creates a hierarchy in OpProc specifications.

This description of the OpProc notation should provide enough of a background to understand readability discussions ahead. For further information regarding OpProc's formal foundation, the reader is referred to [28].

2.2.2 Visualizations

This thesis draws upon another significant area of research in the area of language design. This research focuses on specific features and notations that can make languages more readable and comprehensible, though these results are not necessarily tied to formal specification languages.

Fitter and Green published work in the late 1970's dealing directly with readability in computer languages [9]. While some diagrammatic notations are more expressive and comprehensible than others, it is not necessarily apparent why this is the case. Further, some graphical notations are simply not very good, and not as readable as other conventional notations. For the most part however, diagrams can be extremely successful when designed correctly. Using several examples as well as empirical results, Fitter and Green outline requirements for a readable diagrammatic notation suitable for expressing software system behavior. In doing so, they draw a distinction between parts of a notation that are expressed perceptually (using shapes, colors, layouts, etc.), and parts that are expressed symbolically (or textually).

The first requirement is that information that is encoded perceptually must be relevant. Because there are a finite number of variables that can be used to encode information perceptually (shape, layout, etc.) in a diagram, it becomes extremely important that such information be useful for tasks involving the particular diagram. Otherwise, the perceptual coding is simply a waste of a resource which could be used elsewhere. Several studies have shown that depending on the type of task required,

certain diagrams are more effective than others. For example, Bingham and Davies show by human experimentation that while flowcharts are better at showing procedural flows than are decision tables, it is easier for non-specialists to check and modify decision tables [1]. Both notations can express equivalent information, and certainly both have advantages over each other, but designers must account for the audience and desired tasks to be performed before deciding which perceptual encoding is more appropriate or relevant.

The second requirement for a readable notation is that it should restrict the user to creating forms that are readable and comprehensible. If a notation is too flexible, a notation can be misused and ultimately blamed for allowing unreadable specifications. The need for such a requirement is best shown by example, and Fitter and Green do so with the flowchart. Using an unrestricted flowchart can, "...encourage spaghetti-like programs. In particular, they provide irresistible temptations to jump into the middle of otherwise working construction, violating their preconditions and generating untraceable bugs [25, p. 36]." However, restricting the user to building a flowchart using only a restricted set of building blocks can result in flowcharts which may not be concise, but which will otherwise be readable. Editing a flowchart built from such a restricted set tends to also be simpler than would otherwise be the case with an unrestricted flowchart.

Third, a diagrammatic notation should use redundant recoding to express important information. For example, one means of redundant recoding that is common in software programming is indentation. Two programs may be entirely equivalent in code, but one that is properly indented provides redundant information that is nonetheless easier to read, particularly nestedness. Though not supported with empirical evidence, color seems to be another effective means of redundant recoding. For example, labeling references to a particular variable in a program with a certain color makes it easier to readily see how a variable is affected by a program's execution.

Diagrammatic notations should also reveal the underlying processes they represent, if they are to be readable. Furthermore, images or notations should respond to user manipulation [3]. These requirements are most applicable to interactive systems. In fact, for some applications like air traffic control, interactive displays are almost a

requirement for any sort of representation to capture the dynamics of the processes they represent. The requirement of revelation (i.e. revealing an underlying process) is an important concern when designing database query languages. Fitter and Green describe a sharp division of opinion on how best to express the data-structure and access mechanisms to the user, some preferring a natural language interface, others supporting an interface language designed specifically to reflect a database's underlying structure. At any rate, most parties agree that the ability of a query language to express the underlying organization and access processes of a database is paramount if the language is to be usable.

The final requirement of a readable notation is that it be revisable. Certainly no description of a system's behavior is correct the first time, so the ability of a user to edit a diagram is extremely important. However, this requirement is somewhat contradictory to the earlier requirement that information be redundantly recoded. If a piece of information needs to be revised, but it is redundantly recoded, then there may be additional work involved in revising the diagram. The advantages of revisability vs. redundancy must be weighed when making a decision about a diagrammatic notation.

Of course, following these prescribed requirements cannot guarantee a readable notation. Wright, Brooks, and others have come to the conclusion that psychology and linguistics can only help so much – common sense is just as important in language design, if not more so [34,3]. Of course, what applied psychology can offer that "common sense cannot is quality control: the testing of new designs to see how well they work [9, p. 259]."

Green has also contributed work in human cognition (as it pertains to language understanding) which can be applicable to visualization readability [10]. The field of computer science made several advances in the 1970's and 80's based on simple models of human understanding, which may no longer be appropriate if the field is going to make large advances in the future. Language designers, for example, have sacrificed readability for efficiency, operating under the assumption that computer programmers can become accustomed to anything. While this philosophy can lead to programs that require less typing time, such programs can also be more difficult to debug. Experience

tells us that debugging is an essential and potentially time consuming process for any software endeavor. However, by creating languages that are more expressive to programmers, languages that are based on more sophisticated models of human reasoning, we can reduce debugging and reviewing time significantly.

One of the most prominent contributors to the field of visualization and notation readability is Tufte [31]. Recognizing that "confusion and clutter are failures of design, not attributes of information," Tufte has produced several general principles that can help govern the design, editing, and analysis of information representation [p.53]. These principles are concerned mostly with visual characteristics, such as shapes, colors, and organizations that can communicate information quickly and unambiguously. For example, he warns against the *1 + 1 = 3 effect*, where adding marks to a notation or visualization can activate negative shapes inadvertently. Surrounding words by a box, as shown in figure 17, can activate white space between the box and the word, making the text less readable.

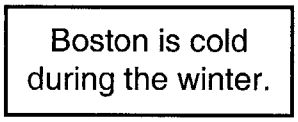


Figure 17. *1 + 1 = 3* Effect of poor visualizations.

Howard has provided a comprehensive survey of other contributors to the field of visualization readability, addressing the readability of material ranging from printed graphics to software systems visualizations [16]. Printed graphics should exhibit certain qualities if they are to effectively transfer information to the reader. For example, a graphic ought to show the data it contains, but do so without distorting what the data has to say. Graphical elements should also serve multiple purposes, conveying several types of information within the same figure. However, such properties should be considered when designing software system visualizations as well.

Event trees, for example, provide assistance with visualizing the communication between components of distributed systems, and are useful for debugging large parallel systems that involve frequent message passing. However, event graphs must deal with the problem of scale, as the graphs can quickly become cluttered and unreadable with the

addition of processes and messages. Analysts have shown that grouping together nodes and messages that provide similar purposes can help reduce the complexity of the graph, while not distorting any information the graph is to display. Visual languages is another means of visualizing system behavior which faces many of the same issues as printed graphics. For example, designers of visual languages must decide exactly what should be graphical in the languages. Certain aspects like control flow might be easier to express graphically, while this type of expression is awkward for arithmetic expressions.

Based on his analysis of the field of visualization readability, Howard presents a list of guidelines that should be observed in visualization design. His taxonomy can be broken down into process, architecture, usability, visualization, memory, and navigation guidelines. While some of these guidelines are only relevant to automated graphic design and manipulation, most emphasize important characteristics such as clarity and consistency, which should be observed in *all* visualization design.

Researchers at Monash University in Victoria, Australia have provided several erroneous principles that are often followed in language design, including "less is more," "more is more," and "violation of expectations" (consistency) [26]. Though these undesirable features are suggested in the context of programming language design, they are certainly relevant (perhaps even more so) to graphics design. Guidelines to creating more learnable, and hence more readable languages are also proposed. One of the most appropriate is that designers should configure to an inexperienced audience. Knowledge of the audience is incredibly important in making languages and visualizations more readable, but is often overlooked in deference to designer preferences. Language designers should also make use of a small, orthogonal set of features. Overlapping features may provide a notational convenience, but the lack of uniqueness may confuse the reader. Including too many features can make a language or visualization difficult to read as well.

2.2.3 Human Experimentation

A final area of research drawn upon in this work deals with human experiment design, specifically within the context of computer science. The use of human

experiments has become an effective means to validate the use of certain software engineering techniques. However, proper methodology must be employed in the design of such experiments not only to generalize results obtained, but to defend work to the research community.

In order to test an experimental hypothesis, a researcher must first create a situation in which the desired behavior can be observed and measured. To create these situations involves a few important issues, namely the selection of subjects, appropriate material, and measures. These issues define an experimental methodology, and will be discussed here [4].

In selecting subjects, there are two criteria that should be considered. One is that they should be a representative set from the test population to ensure that the results can be generalized to the test population (e.g. C++ programmers). However, the subjects should also be as homogenous as possible, to eliminate the possibility of factors such as a subject's background having an effect on the results. These criteria can be difficult to reconcile, especially when the test population is heterogeneous. One solution that experimenters have considered is to recruit subjects from a programming class. This approach can be tempting in that it usually ensures that the subjects' background is homogenous. However, this subject sample is difficult to generalize to the programming population. Several studies have, for example, shown that even a few weeks difference in training can have a significant impact on a subject's preferences and abilities.

Inter-subject variability is a difficult issue to address, and it often controlled by testing a large sample (such as a classroom) and comparing the performances of different subject groups. However, the most attractive method to control subject variability is to design within subject experiments, where a subject is compared only to him/herself. A subject is exposed to all levels of experimentation, and his/her relative performance at each level is analyzed.

In selecting experimental material, the most important issue to consider is that the material taps into the experimental feature being tested. For example, if the use of macros is being tested, it is important that the tested material make use of an appropriate number of macros. However, it must also be representative of a real class of problems.

If experimental material is too contrived or pathological, reviewers will be unlikely to make any conclusions based on the experiment.

Another important issue to consider in material selection is size. Experiments that are based on small programs or specifications do not necessarily lead to scalable results. Experience has shown that building large software systems is not simply a matter of scaling manpower, but requires special methods and techniques. Therefore while a certain mechanism or construct may not have an effect on a small experimental program, it may in practice be influential.

Experimental material must also exhibit a high enough level of difficulty to ensure an appropriately variable level of performance. For example, if the selected material is inherently difficult to read, subjects may perform poorly in all levels of experimentation, making it difficult to make conclusions based on the results.

Selecting an experimental measure may be the most important factor in determining how a particular innovation effects the ease with which a program or specification can be read or written. As Fenton writes:

> In the absence of a suitable measurement system, there is no chance of validating the claims of the formal methods community that their models and theories enhance the quality of software products and improve the cost-effectiveness of software processes [33, p. 216] [6].

One common measure in computer science experimentation involves subject construction of a program. This measure is appropriate if the experiment is testing the ease with which programs or specifications are created with a certain feature or mechanism. Evaluation with this measure can be problematic, especially if the experiment tests the quality of programs produced – this is difficult to judge any way other than subjectively. Another means of evaluating produced programs is to measure the time necessary to construct the program. However, this measure can be problematic for several reasons, including the fact that any time measure should not include irrelevant behavior such as time necessary to understand the directions, as well as time necessary to construct irrelevant parts of the program (i.e. parts that do not involve the tested feature).

Debugging or modifying a program/specification is a useful measure for experiments that involve readability. This measure relies on the assumption that a subject

must understand a program in order to modify it correctly. However, experiment designers must ensure that the requested modifications cannot be made simply (e.g. looking for a particular statement), and that it involves actually understanding the material. Experiment designers have also used memorization and recall tasks to test the comprehensibility of a program/specification [27]. In these tasks, subjects are exposed to a program for a limited amount of time and then asked to reproduce it. Experiment designers should score both functional and literal accuracy. The effectiveness of this measure relies on the assumption that if a program is easy to read and understand, it is easy to learn, an assumption that is well supported. Unfortunately, memorization/recall tasking is only appropriate for small programs/specs. It is obviously unrealistic to ask subjects to reproduce large software systems.

One of the most popular experiment measures in use is question answering. Questions can be open-ended or multiple choice, each with their own attributes. Open ended questions tend to be difficult to score, but can be taken verbally. Multiple choice questions are, of course, easy to score, but are difficult to generate. A final method of measuring program comprehensibility is hand execution, where a subject is asked to trace or simulate execution by hand. One concern with this measure, however, is that the simulation may not require knowledge of a whole program, as execution can be accomplished on a statement basis.

There has recently been research conducted to systematically validate software measurements (e.g. measurement of complexity, readability). Fenton et al. discuss a structural model (shown in figure 18) that serves to describe the parameters involved in measurement, and the relationships between them [17]. With such a model as a basis, they have proposed a framework for validating the use of software measurements. There are two types of validations considered, theoretical and empirical. Theoretical validations verify that the measurement does not violate any of the object's properties, while empirical validations confirm that the actual measurements observed are in line with those predicted from the measurement models. The authors ultimately suggest a list of guidelines to adhere to in measurement selection for computer science experimentation. The most relevant to this work is that if a practitioner is concerned with

a complex property such as readability, then several different measures should be taken in an effort to capture several of its attributes.

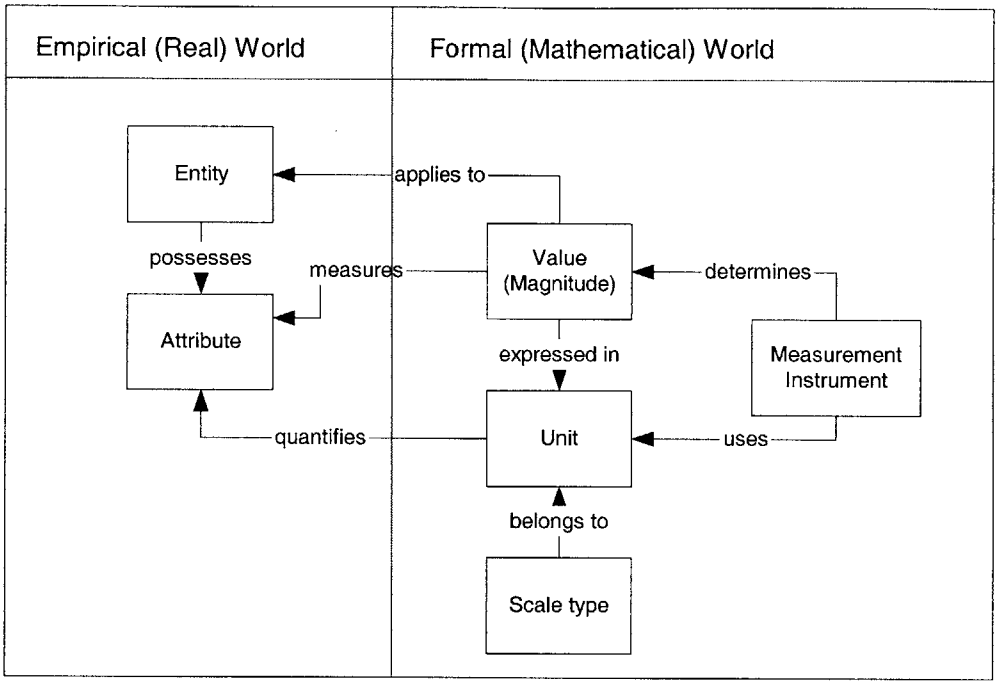


Figure 18. Structural model to define object measurement.

There has also been work involving the establishment of predictive measures, i.e. measures that can predict the occurrence of errors in software and specifications. Based on cognitive psychology experiments regarding the ability of subject to reason about natural language, Vinter et al. have analyzed the ability of software engineers to reason about formal specifications [33]. A human experiment was designed and run to test for the types of reasoning errors made when processing Z specifications. Data collected from this experiment was used to establish a predictive model of error, which can assist with predicting the likelihood of errors being made by a designer using the Z language in a particular task. This experiment bears some semblance to the work described in this thesis. However, Vinter et al. have limited themselves solely to properties of the Z language that are error-prone, rather than determining properties of a class of languages that might be error-prone. Further, this experiment does not account for any subjective feedback which might also be useful in determining those feature of Z that may be difficult to reason about.

Finney and Fedorec performed several experiments regarding the comprehensibility of Z as well. In [7], they set out to determine the effects of comments and meaningful names on the readability of Z specifications. The experiments involved a 2X2 factorial design with four versions of the same specification, each with a different combination of {comments, no comments}, {meaningful names, no meaningful names}. Subjects were taken from several graduate and undergraduate classes, and were randomly given one of the four specifications. They were asked to read the document and answer 3 questions regarding the system specified. They were then given access to every specification, and asked to rank each in terms of comprehensibility. The times required to answer the questions, the scores on the 3 questions, and subjective rankings of the specifications' comprehensibility were recorded. The write-up for this experiment focuses heavily on statistical analysis of data, while underscoring the difficulty of determining a metric for comprehensibility. However, this experiment is not without its shortcomings. The use of time to complete the experiment as an evaluation variable brings with it several complicating factors (as discussed earlier), in addition to the fact that subjects were asked to record their own completion time. Further, asking subjects to answer 3 questions may not provide enough experience with the specification to make any conclusions about its readability. And though alleviating measures were taken in the experiment design, the structure of the specification may have affected the comprehensibility of the specifications. Nonetheless, the results of this work imply that only meaningful naming conventions affect comprehensibility.

In follow-up work, Finney, Fedorec, and Fenton analyzed the effects of structure on the comprehensibility of Z specifications [8]. There are several facets to comprehension, which, as stated before, makes it difficult to test for. The research community has used several means to measure comprehension, some focusing on technical questions and simulations, others focusing on simply reading notations. Finney et al. focus on 4 aspects of comprehension, testing the subjects' ability to perform the following four tasks: finding a relevant part of the specification, understanding the notation, relating the specification to the model, and modifying the specification by writing an extra feature.

Subjects were taken from a class setting, and divided into 3 groups where each group was given one of three different specifications: a monolithic specification (121 lines), a modularized specification, and a highly modularized specification. They were then asked to complete 20 questions testing the four skills listed earlier. Though subjects were taken from a classroom, the results may still be generalized to the software engineering population. Total instruction time was about 40 hours per subject, which is comparable to any formal methods training given to software engineers in industry.

Results of this work show that comprehensibility is in fact improved with modularizing the specification's structure, but that no significant benefits come from breaking down modules that are roughly 20 lines in length. One limitation encountered with generalizing these results, however, is that of scale. The size of the specification tested here is most likely not comparable to those encountered when specifying industrial-size systems. However, the experiment designers argue that common software engineer practice encourages system breakdown into subsystems that are roughly the size of that used in this experiment. There are currently plans to improve the experiment by including timing data, using larger specifications (to assist scalability of results), and by investigating any correlation between a subject's academic background and his/her ability to comprehend Z.

Tenny investigated readability via human experimentation as well [32]. Specifically, he looked at the potential effects of procedure format and comments on the readability of a PL/I program (rather than a formal specification). The experiment was a 3X2 factorial experiment, where each subject was randomly given a program involving a different combination of {inline code, internal procedures, external procedures}, {comments, no comments}. Subjects were then asked 12 short answer / multiple choice questions about the program's behavior. Readability was measured in terms of the speed and accuracy of a subject's responses.

Results of this work implied that the use of procedures does not have a significant effect on readability, and that comments provided the greatest assistance with readability of the program without procedures. It is difficult to generalize these conclusions, however. The tested program, for example, contained between 70-170 lines of code,

depending on the specific features included. This of course, is smaller than most programs encountered in industry. There also does not seem to be any structure or methodology guiding the question selection, which makes it difficult to accept this as an adequate measure of readability. Furthermore, this was a student-based experiment. However, Tenny argues that the students tested were senior, ready to enter the software industry, and that most of the documentation and maintenance positions in industry (i.e. the positions that are most assisted by program readability) are typically filled by newly graduated students.

The work discussed in this thesis draws heavily on previous experimentation design, particularly with respect to measuring readability. However, while there has been a significant amount of research in experiment methodology within a computer science context, the specific goal of determining which factors affect the readability of state-based specification languages has not been addressed. As discussed earlier, there has been work that sought to determine factors that affect the readability of Z specifications, but the number of factors considered (2) is much smaller in scope than that considered here. As well, previous work with Z focuses on one specification language, rather than a class of specification languages as is considered in this thesis. This becomes an important distinction when determining which factors of language design should be tested for readability.

3. Experiment Design

The purpose of this work is to determine those factors that affect the readability of (state-based) formal specifications. This direction of research is fairly innovative and as discussed earlier, has not been addressed by related work. As a first step in this direction, any results obtained are strictly preliminary, and will be used as a starting point for more thorough experimentation in the future.

As mentioned before, this work focuses solely on state-based specification languages, which is just one type of a formal language. The primary motivation behind doing so is that they all employ the same underlying model. When looking at other formal models, there are different issues that arise in expressing their different parts. For example, the different ways to express the parts of a state machine are not the same as those that can express the parts of a set theoretical model. For the sake of a fair comparison, we restricted ourselves to one formal model, and investigated the different ways that this abstract model, the state machine, can be expressed.

The first step of our approach, then, was to survey several state-based specification languages and to determine the distinguishing features of each. What differentiates each of these languages from each other are the ways that they express the parts of the underlying state machine model. After determining exactly what these distinguishing features are, we wanted to test the readability of each in experimentation, to help us identify those features that most affect the readability of state-based specifications.

The languages surveyed were Statecharts, Software Cost Reduction (SCR), Requirements State Machine Language (RSML), SpecTRM-RL, and OpProc Tables. An overview of each was presented earlier. The next section discusses those distinguishing features that we identified as potentially affecting language readability.

3.1 Features Selection

There are several parts of a state machine that must be expressed in a specification language. The features selected can be grouped together according to the parts of the

state machine to which they apply – the state machine (in general), events, and conditions.

3.1.1 Specifying the State Machine

The first factor related to the state machine description is the use of an overview. Modern avionics systems are becoming increasingly complex, which in turn is leading to large, complex specifications. Some languages, such as RSML assist with this complexity by including a graphical overview of its specifications. The graphical overview contains a list of all component state machines in the underlying model, the set of possible values for each state machine, and the allowable transitions within each state machine (there are no triggers included). Like the graphical state machine presented earlier, states in the RSML overview are represented by squares, and transitions between them by arrows. Such an overview encourages developing a system view of the specification. SpecTRM-RL includes a graphical overview as well; however its overviews contain only a list of all component state machines and the set of possible values of each. Experience with RSML showed that presenting the reader with transitions using the arrow notation led to a great deal of confusion when specifying large systems. Using a different notation for transitions may lead to a more readable overview. However, SpecTRM-RL removes this information from its overviews altogether.

In constructing an overview for a complex state machine, there are two issues that should be dealt with. First, one must decide what type of information the overview should contain. For example, the overview can include states, transitions, inputs, etc. Second, the form of the overview must be established. The overviews discussed previously were graphical, but they can also be represented other ways, such as textually or tabularly. We hoped to determine by experiment which types of overviews do in fact assist readability.

The next feature identified was the combination of informal and formal specifications. SpecTRM-RL and OpProc tables incorporate informal requirements into their state machine model. This type of information provides a more complete understanding of a system's behavior, and is obviously required for a complete

specification. However, it may be a source of confusion in understanding the state machine model.

The representation of the state machine itself proved to be a distinguishing feature as well. Op Proc Tables and SCR represented the underlying state machine in a table, whereas Statecharts represented the model graphically, using boxes and arrows. RSML and SpecTRM-RL in essence used text to describe the layout of the state machines. This work tested the readability of equivalent graphical, textual, and tabular state machine representations.

Due to the inherent complexity of modern software systems, most state-based specification languages use superstates, or hierarchies, to provide logical modularizations in the model. The use of hierarchies certainly is a notational simplicity and removes the need to explicitly specify several transitions. For this reason, many have argued that allowing hierarchical specifications is essential if formal methods are to be scalable. It can also help the reader develop a better mental model for the system's behavior than would otherwise be achieved in a flat state machine. However, by not explicitly specifying several transitions, superstates can also lead to confusion regarding execution of the state machine. In the specification languages surveyed, all employ hierarchies, with the exception of SCR.

3.1.2 Specifying Events

One of the most controversial issues in state machine description is the use of internally broadcast events. Statecharts, SCR, and RSML rely on internally broadcast events to order execution of the state machine. However, experience using RSML in a complex specification showed that the use of such events proved to be a source of several errors. SpecTRM-RL then removed internally broadcast events from the language, instead ordering execution based on data dependencies. In this experiment, we hoped to provide empirical evidence regarding the readability of internal events.

Another issue encountered in the use of internal events is *how* their execution should be ordered. Languages like RSML require the designer to explicitly specify the order in which internal events are executed. In addition, RSML can allow different output events to be generated for a single transition. For example, transitioning from a

state A to B may generate one output event in one environment, but a different event in a different environment. This feature provides a great deal of flexibility in describing system behavior, but can make the specification unreadable. Requiring the reader to carefully trace through specification in order to determine execution order can be a truly onerous task. SCR, on the other hand, assigns an event to *every* entity change within the state machine – the event generated depends only on the source and destination of the transition triggered, not on the environment in which it is triggered. So transitioning from A to B will always generate a certain output event, regardless of the environment in which the transition is taken. SCR then adheres to a predetermined scheme to order the execution of these events. This use of internal events provides a less flexible means of specifying a state machine, but may increase the readability of the specification.

3.1.3 Specifying Transitions

The next issue investigated regarding state based specifications is that of perspective. When specifying a state machine, transitions can be organized in one of two ways. They can be organized by source state, where all the transitions out of a certain state are grouped together. One can think of this organization by asking, "If I am in state X, where can I transition to from here?" We refer to this as a going-to perspective. Transitions can also be organized by destination state, where all the transition to a certain state are grouped together, i.e. all the transitions that end in state X are grouped together in the specification. We refer to this organization as a coming-from perspective. Certainly both express equivalent amounts of information. However, we were interested in determining whether one of these perspectives was a more intuitive way for readers to think about state machine behavior. Graphical representations like Statecharts provide both perspectives, which may or may not be an ideal property. Others, like SpecTRM-RL and OpProc tables, use a coming from perspective. RSML and SCR do not restrict the designer to either perspective. In fact, there is no enforced organization of the transitions.

Macros in a state-based specification language function basically as they do in any programming language. They allow us to modularize a piece of logic that can then be referred to solely by name in the specification. This modularization is conducive to

specifying simpler blocks of logic which are theoretically easier to read and understand. Using appropriate naming conventions when specifying macros can also lead to more readable specifications. Previous work with SpecTRM-RL have prompted researchers to conclude that macros are a necessity if formal languages are ever to scale to realistic systems [35]. However, macros also possess a drawback in that it may require the reader to navigate through several parts of the specification to identify specific logic conditions that may affect the state machine. The use of nested macros may confuse the reader when trying to understand how the system behaves. Of course, this problem is not unique to specification languages, but can arise when using programming languages as well. Previous work in software engineering has concluded that the effectiveness of macros is limited by the amount of modularizations used. For example, using a few, non-nested macros can aid comprehension of the system, whereas using several nested macros may not. Whether or not these conclusions apply to state machine specifications is something we chose to investigate in this work. All state based specification languages surveyed employ some form of macros (although they are called *terms* in SCR), with the exception of statecharts.

The final feature selected was the expression of conditions, which of course are used to specify triggers for transitions. SCR and Statecharts use simple propositional logic to specify triggers. This representation is relatively simple and concise, but can be difficult to read when used to express complex conditions. Both RSML and SpecTRM-RL use a tabular notation (the AND/OR table, discussed earlier) to express conditions. Experience using this notation with engineers has shown it not only to be readable, but also easy to learn. OpProc tables also use a tabular notation (which is actually similar to the AND/OR table) to represent conditions. In this experiment, we investigated not only the use of propositional logic and tables to express conditions, but also textual and graphical representations. Textual descriptions are used in most industrial specifications to date, so served as a good baseline for comparison in this experiment. We also considered AND and OR gates as a means of graphically specifying logical conditions. AND and OR gates are familiar to most engineers (and computer scientists), and so seemed an appropriate means of describing the logical triggers of a state machine.

With these features established, we designed a human experiment to determine how each affected a specification's readability. An overview of the experiment follows.

3.2 Experiment Overview

The experiment itself consisted of six (6) parts, one part for each feature tested.[1] Each part of the experiment was run the same. Subjects were presented with 2-4 equivalent specifications, depending on the feature being tested. They were then asked a serious of objective questions about the state machine behavior described by the specifications. Initially, subjects were given access to any/all specifications when answering the first few questions. They were instructed to indicate which specification(s) they used to answer each. We were interested to see whether subjects had an intuitive feeling for which notation or specification they would find to be the most effective or readable. After this preliminary section, subjects were restricted to a particular specification when answering questions until every specification was tested. For example, in the macros experiment, subjects were asked two questions for which they were given access to both the Macro and Flat specifications, then four questions for which they were restricted to the Macro specification, then four more questions for which they were restricted to the Flat specification. Following the objective questions, subjects were asked to give a subjective evaluation of each specification used in that particular part of the experiment.

Subjects were all graduate students in either aeronautics or computer science. Prior to each experiment, the subject's background and familiarity with both fields were ascertained. As mentioned before, there may perhaps be a correlation between a subject's academic background and his/her notation preferences, which makes this an obvious and necessary step to investigate this possibility. Twelve subjects were tested in total, including six computer scientists and six engineers. Statistically, we would like to have a larger subject base, but as this is a preliminary study, we believed that this was an appropriate number of experiments to run.

[1] Three of the factors established above were removed from consideration in the experiment, for reasons discussed later.

As far as training, subjects were given a simple introduction to state machines a week before their experiments. The document simply familiarized them with basic state machine terminology such as "states," "triggers," and "transitions." This introduction provides a subject with no experience with state machines enough information to complete the experiment. Furthermore, a practitioner was present throughout each experiment to explain the directions for each part of the experiment, including how to read the notations, as well as to answer the subjects' questions.

3.3 Experiment Material

For each feature tested, we decided upon a specific system with which the subject would work. We developed several specifications for each system used in the experiment. The specifications for a particular system were entirely equivalent, and differed only with respect to the particular feature being tested.

Deciding appropriate systems to use in the experiment is an important task, which can directly affect the credibility of any conclusions reached as a result of the work. There are several issues that were considered in designing material for each part of the experiment. First, we wanted the systems specified to be taken from several aeronautical applications. Aeronautics is largely a safety-critical field, and one that could benefit greatly from readable, reviewable specifications. So, we designed the experiment in an aeronautics environment, though the results should be applicable to any software specification.[2] More importantly, the systems selected were real systems. We did not want to test systems to be contrived or pathological, but rather we hope to use this experiment as a practical experience. Using real systems should help us ultimately create readable specifications that can actually be used in an industrial setting.

In developing this work, we were concerned with the readability of large-scale system specifications. Specification language features can become much more influential when dealing with large systems. However, we restricted the size of the systems used for a couple of reasons. First, we wanted to minimize the duration of the experiment. Several features were being tested in this experiment, and we recognized that duration

[2] Both space and aviation systems were used in the experiment

may affect a subject's performance. Using large system specifications can result in a significant amount of time necessary for a subject to become familiar with and be able to read them. We also limited the size of the test systems so as to minimize potential sources of experimental error. As discussed in related work, several factors of a system specification can affect its readability, including the way a specification is modularized, fonts used, and naming conventions. As the size of the system is increased, there is a greater potential for such factors to affect a specification's readability, in addition to the specific feature being tested. By restricting the size of the specifications used, we hoped to minimize any potential effect that external factors may have on the subject's performance.

Another concern in designing the experiment material is that the notations used be generic, so that it not require much training or foreknowledge of the subject. We also felt it important to vary the notations used, so that a subject's preferences would not be affected over the duration of the experiment. For example, if we described triggers using propositional logic whenever possible, the subject might become biased against other notations, which would affect his/her performance on the conditions part of the experiment.

The overviews and formal/informal requirements were ultimately not tested in the experiment. In designing the overview experiment, we decided that it would be unlikely that an overview would ever detract from a specification's readability. We contemplated testing how the inclusion of different types of information affected an overviews readability (for example, showing possible states and transitions vs. showing possible states only), but found that the readability of such overviews depended on the types of questions being asked. This issue did not seem particularly interesting for this work, and so was not pursued further. The inclusion of formal and informal requirements in a formal model was not tested due to our belief that any complete specification must contain informal requirements (such as timing requirements) as well, whether or not this comes to the detriment of the specification's readability.

The experiment material selected and the notations used will be discussed in detail in the next section

3.3.1 Representation

This experiment was designed to test the following (null) hypothesis:

Hypothesis 1: The representation of a state machine model does not affect its readability.

The system selected for the state machine representations part of the experiment was the control software for the HETE (High Energy Transient Explorer) satellite. "The purpose of the HETE mission is to study gamma ray bursts. Gamma ray bursts are high-energy transients in the gamma ray portion of the electromagnetic spectrum that seem to be isotropically distributed in the sky. These transient energy bursts range in duration anywhere from a millisecond to a few hundreds of seconds, and they involve a high amount of energy. In addition to capturing spectral information about these bursts, the craft also relays position data about bursts as they occur to ground stations and other satellites so that they can also study the phenomena [24]."

The control system software can be modeled as a simple state machine with ten (10) states, each representing a different mode of behavior. The specifications for the system contain the different modes of behavior, as well as the conditions under which the system transitions between them. They do not contain any information regarding how the system operates in each mode. The state machine model for HETE was appropriate in that it did not contain an unreasonable number of states or transitions, nor was the transition logic complex. Therefore we were able to express the state machine using the tested representations, without drawing too much attention to any particular part of the state machine. If we had chosen a state machine with complex triggers, for example, the readability of the specifications would likely hinge on the representations of the conditions, rather than the representation of the state machine.

This part of the experiment presents a textual, graphical, and tabular representation of a state machine model for the HETE control system. The textual specification is written in straightforward English text. For example, figure 19 shows one part of the specification. Simple mathematical symbols such as >, <, and ≠ were used in the textual specification. The transitions in the specification were organized by source state. For example, figure 19 contains all of the transitions out of Wait Mode.

Wait Mode

Transition from Wait Mode to Wait Mode if the time that the system has been in the Wait Mode is < Wait Mode Delay.

Transition from Wait Mode to Detumble Mode if the time that the system has been in the Wait Mode is ≥ the Wait Mode Delay.

Figure 19. Sample textual specification.

The tabular specification for the control system uses a fairly generic notation, resembling OpProc Tables. There are of course several ways to organize a tabular representation of a state machine. However, it was important to have the entire state machine (states, transitions, and triggering conditions) rely solely on the setup of the table, rather than other notations. Each column in the table describes a transition, i.e. a current mode, and destination mode, and a trigger for the transition from the current to the destination mode. The source mode is listed in the top row, the destination mode in the bottom row. Inputs to the system are listed in the far left column. Triggers for transitions are described as a conjunction of various values of these inputs. So, a transition will be triggered if every element in the corresponding column is true. Figure 20 shows an example of such a table. The system will transition from Performance Mode to Economy Mode if the Pilot Requested Mode is economy and the flight phase is Cruise.

Current Mode	Performance
Altitude	
Thrust Limit	
Pilot Requested Mode	Economy
Engine Out	
Flight Phase	Cruise
Destination Mode	Economy

Figure 20. Sample specification table.

Blank boxes denote "Don't Cares" -- the input can take on any value. In this example, the transition from Performance to Economy mode does not rely on the Altitude, Thrust Limit, or Engine Out inputs, so they can assume any value. As in the textual specification, transitions in the tabular specification were grouped together according to source state. We did not want to vary the organization of the transitions, as this might affect a subject's performance.

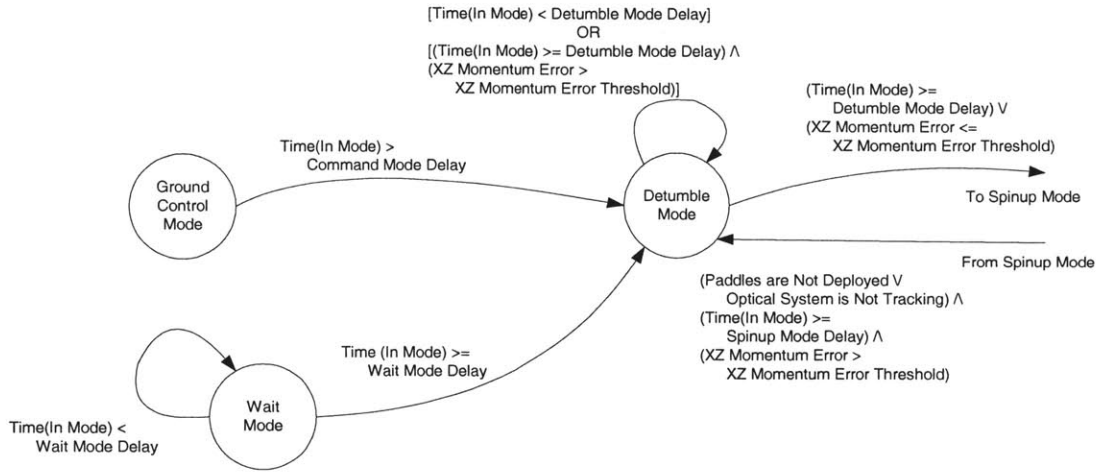


Figure 21. Sample graphical specification.

The graphical representation of the state system is fairly standard, where circles represent states, and arrows denote transitions between them. A part of this specification is shown in figure 21. This graphical notation has a small advantage above the other in that it conveniently provides both a going-to and a coming-from perspective. When looking at a certain state S, it is easy to see not only those transitions that have S as a source, but also those transitions that have S as a destination. The triggering conditions are written using simple propositional logic, and are written directly on the transition arrow. This, of course, is one disadvantage of the graphical state machine. Expressing conditions using a form other than text or logic is difficult due to spatial considerations. The graphical specification spans 3 pages (another disadvantage of graphical state machines), and can be found in the appendix, along with all specifications used in the experiment.

3.3.2 Conditions

This part of the experiment was designed to test the following (null) hypothesis:

Hypothesis 2: The representation of trigger conditions does not affect its readability.

The system chosen for this part of the experiment is a simple Speed Mode indicator, which operates as part of a flight management system (FMS). The Climb FMS Speed Mode describes the mode to which the FMS should transition, and does so based on the environment of the system. This simplified system can be modeled as a single state machine with four states: Default, Economy, Max Climb, and Edit. These states and the conditions that trigger transitions between them are described in each specification. This indicator served as a suitable example for this part of the experiment. It has a small number of states and transitions, but the conditions themselves are quite complex.

In this part of the experiment, four different notations for the expression of conditions were tested – textual, graphical, tabular, and logical notations. The triggering conditions are broken down into a disjunction of conjunctions, so that they can be expressed in a similar format, regardless of the notation used. This approach may mask some benefits obtained by using certain notations, but should also minimize the effect of structure on the readability of the specifications. The text expression reads as straightforward English text, similar to that seen in the representations experiment. For example, one part of the textual state machine reads:

> The Climb FMS Speed Mode shall be the **Default** if any of the following scenarios are true:
>
> 1. the Flight Phase transitions to Done
> 2. the Flight Phase transitions from Takeoff to Descent

The graphical notation makes use of AND/OR gates to express conditions. There are several different ways to express conditions graphically. However, the AND/OR gate is a notation with which most engineers are familiar. Figure 22 shows a sample taken from the graphical speed mode specification. If the final gate (reading from left to right)

evaluates to true, then the speed mode will be *Economy*. The graphical specification was the lengthiest, due to the spatial layout required by the AND/OR gate notation.

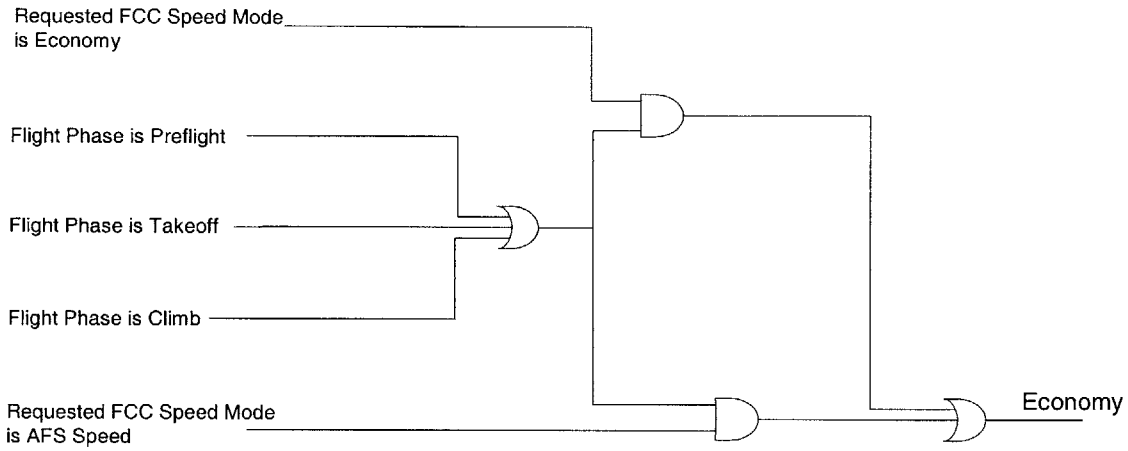


Figure 22. Sample graphical specification for trigger conditions.

Economy =

```
((Requested FCC Speed Mode = Economy) ∧
 ((Flight Phase is Preflight) ∨
  (Flight Phase is Takeoff) ∨ (Flight Phase is Climb))) ∨
((Requested FCC Speed Mode = AFS Speed) ∧
 ((Flight Phase is Preflight) ∨
. . . . .
```

Figure 23. Sample logical specification for trigger conditions.

The logical specification uses simple propositional logic to express conditions. Figure 23 shows a sample taken from the logical speed mode specification. One problem common to parenthetical notations is that they are difficult to decompose, i.e. to see how the parentheses line up. For this reason, a standard size font (courier) was used, and lines of the specification were indented and aligned to make the AND/OR structure of the logic more readable. The logical specification was nonetheless the most concise of the four tested. The tabular specification uses AND/OR tables (described earlier) to express trigger conditions. The AND/OR table is, of course, just one type of table that can be used here. Figure 24 shows a sample taken from the tabular speed mode specification.

= Economy **IF**

Requested FCC Speed Mode = Economy	T	T	T	*	*	*	*	*
Flight Phase = Preflight	T	*	*	T	*	*	*	*
Flight Phase = Takeoff	*	T	*	*	T	*	T	*
Flight Phase = Climb	*	*	T	*	*	T	*	T
Requested FCC Speed Mode = AFS Speed	*	*	*	T	T	T	*	*
Requested Climb Speed Mode	*	*	*	*	*	*	T	T

Figure 24. Sample tabular specification for trigger conditions.

3.3.3 Macros

This part of the experiment was designed to test the following (null) hypothesis:

Hypothesis 3: The use of macros in the representation of trigger conditions does not affect their readability.

The system chosen for this part of the experiment was a vertical guidance reference altitude. This subsystem operates as part of an FMS, and computes the altitude reference used in guidance and control functions. The reference altitude is based on the aircraft's current flight phase, clearance altitude, target altitude and conflict altitude. The computation of the vertical guidance reference altitude can be modeled as a single, fully-connected state machine with roughly 10 states. Like the Climb FMS Speed Mode state machine, this state machine has a relatively small number of state, but contains complex trigger conditions which may warrant the use of macros. A system with simple triggers would render macros useless, and so would be inappropriate for this part of the experiment.

Subjects were given two specifications for the reference altitude, one with macros, and one without. Both specifications were written using AND/OR tables. We did not want the readability of the specifications to be hampered by logical notations. AND/OR tables have been proven to be easily readable by an engineering audience with a minimal amount of training, so seemed an appropriate mechanism to specify triggering conditions. One issue encountered with placing macros in the specification is deciding exactly what information should be modularized. We did not use any formal principle

when modularizing logic in the trigger conditions. The most complex triggers were identified, and then simplified by subjectively assigning macros. Small pieces of logic that were referenced several times were also modularized. The macro specification allows one level of nested macros in some places, which seemed appropriate. Studies have shown that several layers of nesting can create confusion, but large system specifications could remain quite complex if nesting were altogether prohibited.

= Climb Conflict Altitude **IF**

Engine Out is not engaged	T
Climb Conflict Situation	T
Climbing	T

with macros

= Climb Conflict Altitude **IF**

Engine Out is not engaged	T	T	T	T	T	T	T	T
Flight Phase is Takeoff	*	*	*	*	T	T	T	T
Flight Phase is Climb	T	T	T	T	*	*	*	*
Clearance Altitude < Aircraft Altitude - 250 ft.	T	*	T	*	T	*	T	*
Climb Target Altitude < Aircraft Altitude - 250 ft.	*	T	*	T	*	T	*	*
Vertical Guidance Type is Profile	*	*	T	T	*	*	T	T
Vertical Guidance Type is Airmass PROF	T	T	*	*	T	T	*	*
FCC Autopilot is engaged	T	T	T	T	T	T	T	T

without macros

Figure 25. Sample reference altitude specification with/without macros

Another factor of the macro specification that may affect readability is naming convention. We named macros in such a way that made semantic sense, according to the function of the logic it contained (For example, we named a macro *Cruise Conflict Situation*, rather than *Macro A*). References to macros were denoted using italics. For example, figure 25 shows part of the vertical guidance reference altitude specification using macros, and the corresponding part of the flat specification. The table with macros refers to *Climb Conflict Situation* and *Climbing*, which are macros defined by separate AND/OR tables earlier in the specification.

3.3.4 Internal Events

This part of the experiment was designed to test the following (null) hypothesis:

Hypothesis 4: The use of internally broadcast events in a state machine description does not affect its readability.

The system chosen for this part of the experiment is a simple altitude switch. The altitude switch device operates onboard an aircraft and uses several sensors to determine whether the aircraft altitude is above or below certain thresholds, and then sends a signal to a controlled device. The system can be modeled using seven simple state machines, operating in parallel. It was important to use multiple state machines in the system model to make practical use of internal events for communication between state machines. On the other hand, controlling the scale of the system was equally important. Presenting the subject with numerous state machines and extensive internal events can make this part of the experiment particularly time consuming. Our original state machine model for the altitude switch contained 9 state machines, which proved to be demanding during pretesting. Ideally, we would have liked to test the readability of internal events in large specifications, as previous work with RSML suggests that this feature is extremely problematic for such tasks. However, minimizing the duration of the experiment was equally important.

Subjects were presented with two specifications for the altitude switch, one with internal events, and one without internal events. Both specifications are specified using generic tables. This representation may or may not be the most readable, but again we attempted to vary the notations used as much as possible. The tabular notation used is also very concise, which was important given the size of the system specified.

State	**Trigger**
Unknown	Startup OR Controls Reset OR [(**Analog Altimeter** = Unknown) AND (**Digital Altimeter** = Unknown)]
Below Threshold	[(**Analog Altimeter** = Valid) AND (Analog Altimeter Value < 2000)] OR [(**Digital Altimeter** = Valid) AND (Digital Altimeter Value < 2000)]

Figure 26. Sample eventless specification.

A part of the eventless specification can be found in figure 26. Each state machine is described by a separate table. The first column lists the possible states of the state machine, and the next column describes the triggers, i.e. the conditions under which the state machine will transition to each state. The state machines are fully-connected, and conditions are described using simple ANDs and ORs. There is no explicit communication between state machines by events. Rather, execution is determined based on data dependencies, as in SpecTRM-RL.

State	**Trigger** (Event)	(Condition)	**Output Event**
Unknown	Startup, OR Controls Reset		ALTITUDE UNKNOWN
	ANALOG ALT UNKNOWN	(**Digital Altimeter** = Unknown)	
	DIGITAL ALT UNKNOWN	(**Analog Altimeter** = Unknown)	
Below Threshold	ANALOG ALT VALID	Analog Altimeter Value < 2000	ALTITUDE BELOW
	DIGITAL ALT VALID	Digital Altimeter Value < 2000	

Figure 27. Sample specification with internal events.

Part of the specification with internal events is shown in figure 27. The table is slightly more complicated, due to the additional amount of information contained. The trigger now can contain an internal event and/or condition. There is also an Output Event column. When a transition is taken, an output event can optionally be generated, which is visible to other state machines in the system. For example, in figure 27 we see that if the Altitude state machine transitions to state *Below Threshold*, and output event ALTITUDE BELOW will be generated which can serve as a trigger event for other transitions. All internal events are written in caps, to assist with readability.

The altitude switch system is the largest state machine used in this experiment, and can be difficult to understand. However, we were hesitant to simplify the system any more, as it would make the use of internal events irrelevant. Therefore, subjects were given a graphical overview of the system to use in this part of the experiment. The overview contained the names of all seven state machines, each of their possible states, as well as inputs and outputs of the system.

3.3.5 Hierarchies

This part of the experiment was designed to test the following (null) hypothesis:

Hypothesis 5: The use of a hierarchical state machine does not affect its readability.

The system chosen for this part of the experiment was a simple digital timer. The timer keeps track of the current time, signals an alarm, and possess a stopwatch timer mode. The system can be modeled as a single state machine with 18 states. Conditions that trigger conditions in the system are simple (e.g. button a is pressed). The timer seemed an appropriate system here as there are enough states to justify using a hierarchy in some places, and readability is not impeded by complex triggers.

The system was specified as a graphical state machine which should be a familiar representation to subjects, and therefore not one that affects readability. Subjects were given two different specifications for the timer, a flat state machine, and a hierarchical state machine. Both can be found in the appendix. We contemplated providing subjects with a third state machine that was semi-hierarchical in order to test the effects of increasing modularity. However, this seemed inappropriate for a system this small. As well, we were hesitant to use a larger system due to concerns over the duration of the experiment.

3.3.6 Perspective

This part of the experiment was designed to test the following (null) hypothesis:

Hypothesis 6: The perspective (going-to vs. coming-from) used in a state machine description does not affect its readability.

The system chosen to test the readability of perspectives was the HETE control software, discussed earlier in the representations experiment. We expected subjects to be familiar with the general functionality of the system (which would simplify this part of the experiment), but did not feel that this would affect their performance here.

Subjects were given two specifications for the HETE control software, one using a coming-from perspective, the other a going-to perspective. Both specifications use the generic tabular notation described in the representations experiment earlier. The only

difference between the two is the organization of the transitions. The coming-from specification groups together transitions based on destination state, while the going-to specification groups transitions together based on source state.

3.4 QUESTION DESIGN

To measure readability, we were looking for indications that the subject is able to read and understand material in the specifications. Several approaches to measuring readability have been proposed in previous work, some relying on specific technical questions, others on general questions about a system's functionality. In this work, we followed the approach taken by Finney et al. [8] and Brooks [4].

As no single measurement can capture all aspects of readability, we considered both objective and subjective evaluations. Objective questions were designed to test for four different aspects of readability. They are listed below, in order of increasing difficulty:

Finding a relevant part of the specification
For example, one sample question of this type might be, "Where in the specification is the trigger specified for a transition from the Cruise to Descent state."

Understanding the notation
For example, one sample question of this type might be, "What does line 6 of the specification say?"

Relating the specification to the model
For example, one sample question of this type might be, "What will the output of the system be if the Altitude input is 1000 ft.?"

Modifying the specification
For example, one sample question of this type might be, "What changes need to be made to the specification if the transition `Reorient mode to Spinup Mode when condition C occurs' is added to the state machine?"

The number of questions of each type that were asked was relatively balanced. We did not consider one type of question to be more important than another.

A lot of time was devoted to designing the objective questions for the various parts of the experiment. Our biggest concern was that the questions asked involve

realistic tasks, i.e. tasks that would be encountered during the review and modification of real system specifications. We were aware that asking unnecessary questions would only result in a longer duration for the experiment. We also took into account the potential diversity in our subjects' backgrounds, recognizing that those that had never seen a state machine before may require more time to complete the experiment than estimated. This concern was validated during pretesting. Therefore, subjects were asked between four and five questions about each specification, which allowed us to test the desired aspects of readability, without requiring an inordinate amount of time for most subjects to complete.

Editing the objective questions asked proved to be a time-consuming task. Again, we removed any question that did not involve a realistic engineering task. For example, the question "How many binary state machines are described in the specification?" Answering this question certainly involves reading and understanding the notation used in the specification, but the task is not a realistic one. Several questions were removed or edited simply because they were too difficult or tedious, particularly simulation questions. We found during pretesting that questions of the form, "To which state will the system transition in the following environment?" took an inordinate amount of time for subjects to answer. These are, of course, important types of questions asked of specifications in practice. However, we tried to control the experiment's duration whenever possible.

Another issue encountered while editing the experiment's questions was consistency throughout the various parts of the experiment. For example, when subjects are given three different specifications testing the readability of a certain feature, we would like the types of questions asked of the three specifications to be as similar as possible in terms of format and skills involved to answer. Obviously if more difficult questions were asked of one specification, that would not only affect the subject's objective performance in the experiment, but would likely affect his/her subjective evaluation of the specification's readability as well.

One point that should be made is that no time measurements were taken during the experiment. As discussed earlier, there are several (often overlooked) difficulties and

inadequacies involved with using time as a recorded variable. Furthermore, we did not want stress to play any part in a subject's performance. We wanted subjects to be able to work through every question in the experiment, so that their subjective evaluations would be as complete as possible (subjective evaluations will be discussed shortly). If subjects were given a time limit, they may not have been able to attempt every question. As well, they may have made errors in responding to questions that they would not have otherwise made. Each part of the experiment contained an average of 12 objective questions and was designed to take between 20-25 minutes long, bringing the total length of an experiment with 6 parts to roughly 2.5 hours. Subjects were also encouraged (but not forced) to skip questions that required more than 90 seconds to answer. Skipping difficult questions provides useful information as well. For example, if it takes longer than 90 seconds to answer a certain type of question using a certain specification, we may be able to make a judgment about the specification's readability.

After each part of the experiment, subjects were asked a set of subjective questions regarding their experience using the specifications. In some respects, this measurement is more important than the subjects' objective performance. Readability is a complex property which is difficult if not impossible to measure objectively. Subjects were asked to rank the specifications used for each part of the experiment in terms of readability, and then in terms of ease of editing. They were also asked to identify advantages and disadvantages they found using each specification. Subjective responses were taken verbally, which we feel is important for a couple of reasons. First, subject tend to be more expressive when communicating orally, rather than in writing. We hoped that this would lead to more insightful responses. Second, asking the experiment practitioner to record subjects' subjective responses would lessen the workload of the subjects, and hopefully help reduce the total duration of the experiment.

The experiment as administered, including test specifications and objective and subjective questions asked of the subject can be found in the appendix.

4. RESULTS

There were two parts of the results analysis – objective and subjective. The objective questions for all subjects were graded by the same person to ensure consistency in the analysis. Questions were graded as either correct, partially correct, or incorrect. We did not refine the grading process anymore than this, as it did not seem likely that it would affect our conclusions. This grading system is by no means ideal, however. A subject may answer a question incorrectly, but it is unclear to the grader whether the subject made a careless error, or whether he/she did not understand the specification(s). The only way to clear up this ambiguity would be to talk with each subject about the thought process involved in answering each question, which was not feasible in this experiment.

Another problem with this grading system is that although subjects were encouraged to skip questions that required more than ~90 seconds to answer, subjects often worked on a question for several minutes due to the fact that there was no enforced time limit. This situation is not reflected in our grading system. If a subject worked on a problem for 5 minutes, but answered it correctly, it would be recorded just as any correct answer.

Because only twelve subjects were tested, only strong patterns were noted. No conclusions can be made from a weak pattern occurring in such a small sample base.

In general, the experiment was run with few problems. For example, a few subjects required additional explanation regarding the use of internal events after reading the introduction to state machines provided to them. In instances like this, it proved worthwhile to have a practitioner present throughout the experiment. Otherwise, the training of the subjects appeared to be sufficient to complete the experiment.

The only major problem with the experiment was its duration. As discussed earlier, we took great care to control the duration of the experiment and anticipated, based on pretesting, that the experiment would be completed by each subject in roughly 2.5 hours. However, in practice the experiment took anywhere between 2.75 and 4 hours for each subject to complete. Experiments of this length can of course affect a subject's

performance. Subjects were offered short breaks (~10 minutes) after each part of the experiment completed to lessen the effects of its duration, if necessary. More design and pretesting could better alleviate this problem. Again, we considered enforcing a time limit as well, but rejected that idea for reasons cited earlier.

Though performance time was not officially measured, it was interesting to note that in general, those with a computer science background performed much faster than those with an aeronautics background, and with comparable accuracy. Computer science students finished the experiment in approximately 3 hours, and aeronautics students in almost 4 hours. This observation was contrary to our expectations. We expected that although aerospace engineers were not likely to be familiar with state machines, they would be familiar with the types of aerospace systems specified in the experiment, and would hence perform faster than computer scientists. However, it seems as though familiarity with state machines is a much more influential factor in reading formal specification than is familiarity with the systems themselves. This observation may help explain the lack of widespread adoption of formal methods among aerospace industries – aerospace engineers are not accustomed to using state machines.

Speaking with one of the computer scientists after the experiment helped offer another explanation for the difference in performance times. He said that he actually enjoyed taking the experiment because he found that a lot of the questions asked reminded him of computer science exams, that the skills used in the experiment were the same as those of his education. Computer science exams often involve asking a subject to trace through a system specification, or to answer questions about a system's behavior given a specification, which are very similar to the types of questions asked in this experiment. Aeronautics students are rarely asked to answer questions about aerospace system specifications. However, these types of tasks and skills are likely going to become more important as automation assumes a larger part in the field, and so it may benefit students to be exposed to them during their education.

Following is a detailed review of the results obtained for each part of the experiment.

4.1 Representation

As shown in figure 28, subjects consistently performed very well on the objective questions, which made it very difficult to draw conclusions about readability. Most of the mistakes made by those with a computer science background occurred when dealing with the tabular specification (questions 11-14), whereas every aeronautical student answered these questions correctly. The computer science subjects answered an average of 3.3 of the 4 tabular specification questions correctly, where the aeronautical students answered all 4 correctly. Using a two-tail t-test, we were able to show a statistically significant difference between these means at the 97.5% confidence level, which implies that aerospace engineers may find tables easier to use than computer scientists.

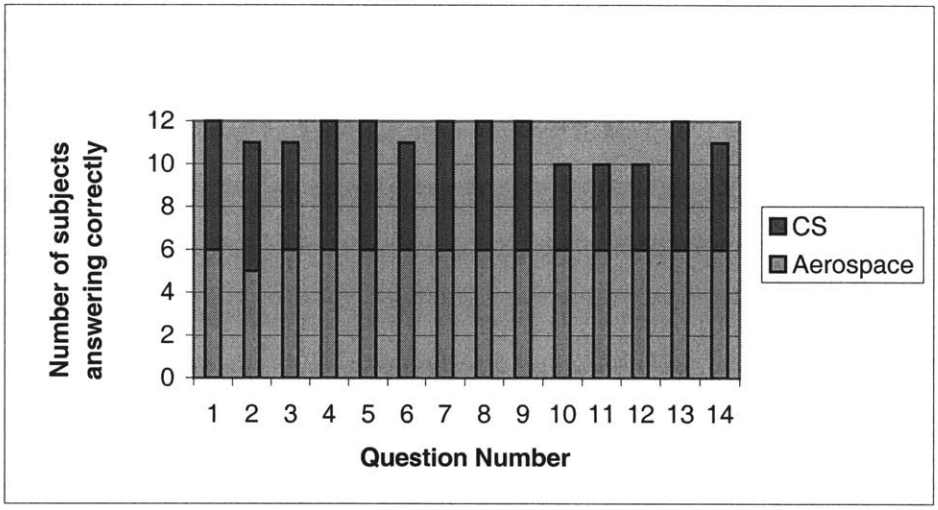


Figure 28. Histogram of Subject Performance for Representation Experiment.

For the first two questions in this part of the experiment, subjects were given access to all specifications. Without exception, every subject chose to use the table for the first question, and all but two subjects chose to use the table on the second question (as a matter of note, only one subject answered the second question incorrectly, and this subject was one of the two that did *not* use the table). Though the particular table used in this experiment may be more readable than others, it appears as though subjects prefer to use a tabular representation of a state machine, which is in agreement with their subjective rankings.

Subjects found that the graphical specification was useful for obtaining a high-level understanding of the system, but that it became cumbersome when asked to answer

questions about triggers for transitions. This difficulty may be dependent on the propositional logic used to represent triggers, but is likely a drawback to graphs in general. Graphical state machines are only practical when concise trigger representations (like logic) are used. The tabular specification was better suited for answering questions about specific transitions, such as whether two transitions are consistent. Though easy to read, subjects (with two exceptions) found the textual specification very difficult to use in the experiment.

According to subjective rankings, subjects on average preferred the tabular specification, followed by the graphical, and then the textual. However, the ranking of the tabular and graphical specifications differed slightly depending on background. Five out of 6 computer scientists ranked the tabular specification the most readable, whereas only 2 aeronautics students did so. Rather, 3 out of the 6 aeronautical students ranked the graphical specification the most readable. This difference between computer scientists and aeronautics students does not appear significant, however.

Ranking the textual specification as the least readable was consistent across subject background.[1] This observation is not altogether surprising. However, it is significant in that most specifications used today are specified textually.

4.2 Conditions

As shown in figure 29, subjects generally performed very well on the objective portion of the conditions experiment, with no discernable pattern existing among the types of questions answered incorrectly. Two of the questions drew a few more errors than the others. The first, question 13, dealt with the AND/OR gate, and asked the subject to trace through a part of the specification to determine whether the FMS Mode could be Edit in a certain environment. One subject answered this question incorrectly, and two others skipped it, presumably because it took longer than 90 seconds to answer. These results suggest that the AND/OR gate notation may be error prone, though the evidence is not overwhelming. Four subjects answered question 18 incorrectly, which asked them to edit the propositional logic specification. These errors were mainly due to

[1] It is important to note that for the sake of subjective rankings, readability is defined with respect to how usable a specification was when answering questions in the experiment. The textual specification may technically be easy to read, in that it is simple English prose, and does not make use of any special notation.

mismatched parentheses and brackets, which of course is a difficulty encountered when expressing complex triggers using propositional logic.

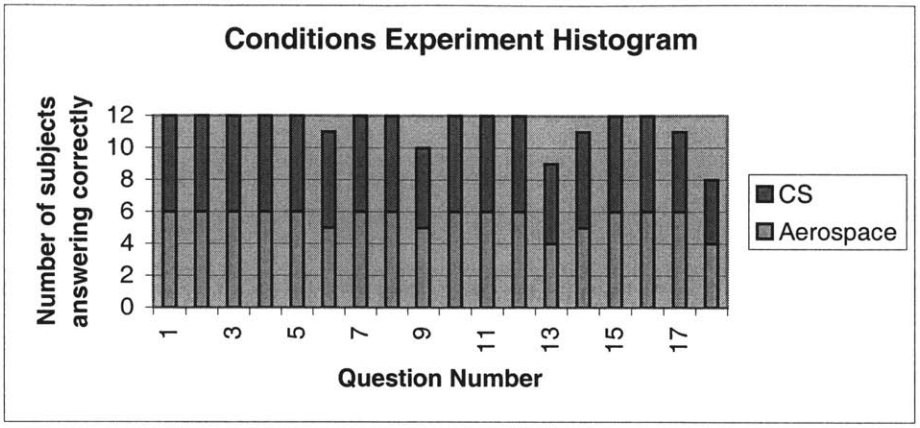


Figure 29. Histogram of Subject Performance for Condition Experiment.

Subjects were given access to any specification for the first two questions. Every aerospace student chose to use the tabular representation for these questions, which supports Leveson's theory that the AND/OR table is easy for engineers to use. It was somewhat surprising that no engineer chose to use the graphical specification. We hypothesized that engineers would be familiar with AND/OR gates and so would prefer to use that notation. The computer science students used the textual, tabular, and logical specifications for these questions, making it difficult to draw conclusions.

Most subjects found the tabular representation concise and easy to use in the experiment, though several complained that the large number of "don't cares" often make the table confusing. One potential solution is to use a blank square to denote "don't care," rather than a dot. Interestingly enough, every computer scientist commented on the difficulty of using the graphical specification. Feelings about the graphical specification were mixed among the aeronautical students. This observation may be explained by the fact that computer scientists do not encounter AND/OR gates as often as students of a traditional engineering curriculum. Six subjects (computer science and aeronautics) expressed difficulty parsing the propositional logic specifications, though this may be assisted by using different parentheses or brackets.

Several subjects made comments regarding the difficulty of using the textual specification. Two subjects in particular admitted that when answering the questions for

which they were restricted to using the textual specification, they actually gave up and used a different specification to answer them. For the most part, however, subjects found the readability of the textual specification to be adequate. One subject remarked that he found the textual specification to be the most useful when answering the questions, simply because the questions were given in English text as well. It is interesting then that so many other subjects found non-textual representations (e.g. tabular) so readable, despite the fact that questions were presented textually.

Subjects were asked whether they preferred the simplicity of a single notation, or whether it was worthwhile to be provided with multiple representations for trigger conditions. Ten of the twelve subjects felt that it is worthwhile to provide an analyst with two different representations for trigger conditions. All ten of these subjects felt that one of these should be the tabular specification, while choices regarding the second representation were mixed between the textual and graphical specification. As a matter of note, the two subjects that preferred the simplicity of a single representation felt that it should be tabular.

		Rankings for Readability			
Background	Subject	Text	Table	Graph	Logic
Aeronautics	1	3	1	4	2
	2	2	1	3	4
	3	3	1	2	4
	4	3	1	2	4
	5	2	1	3	4
	6	1	2	4	3
Computer Science	7	1	1	4	3
	8	2	1	3	4
	9	2	3	1	4
	10	2	1	3	4
	11	4	1	3	2
	12	1	4	3	2
Aeronautics Average		2.33333	1.16667	3	3.5
Computer Science Average		2	1.83333	2.8333	3.16667
Overall Average		2.16667	1.5	2.9167	3.33333

Table 1. Subjective Rankings for Conditions Experiment

As far as subjective rankings shown in table 1, subjects on average found the tabular specification to be the most readable, followed by the textual, then graphical, and finally logical. In fact 9 of the 12 subjects ranked the tabular specification the most

readable. These rankings were consistent among computer science and aerospace students. However, the rankings were slightly different with respect to ease of editing. Subjects on average found the tabular specification to be the easiest to edit, followed by the graphical, textual, and logical. Based on both objective and subjective evaluations, the tabular representation appears to be the most readable of those tested, while the logical proved difficult to understand when used to express complex behavior.

4.3 Macros

Subjects generally performed very well on the objective portion of the macros experiment, as shown in figure 30. Question 9, however, drew several errors, with only 5 of the 12 answering it correctly (those five included both aerospace and computer science students). This question restricted subjects to the macro specification, and asked them to execute the state machine by hand in a given environment. Navigating through various macros may have proved difficult here. A comparable question (question 5) was asked of the flat specification, which 11 of the 12 answered correctly. These results imply that it may be difficult to assess how specific conditions affect system behavior using a specification that employs macros.

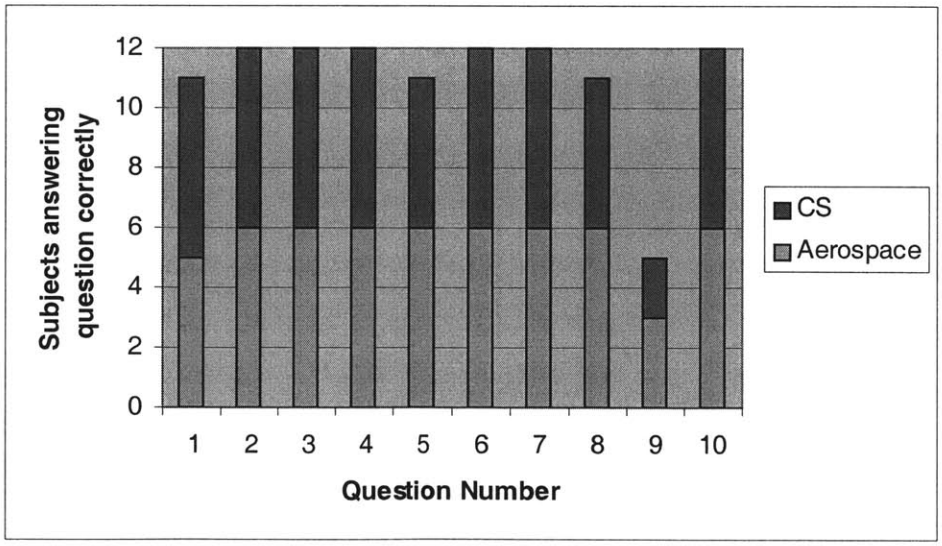


Figure 30. Histogram of Subject Performance for Macros Experiment.

Subjects were given access to both the flat and the modularized specification for the first two questions. Eleven of the 12 subjects chose to use the flat specification. While experience has shown that macros are almost essential when developing a specification, it may be the case that they actually make a specification less readable.

This hypothesis was echoed in the subjective feedback given in the experiment. Subjects consistently felt that macros were beneficial when specifying complex systems, leading to less cluttered and more compact logical expressions. They also noted that macros assist reuse, and that with proper naming conventions, they can make logical expressions easy to read. However, subjects also commented that macros often made it difficult to navigate complex specifications, often requiring lots of flipping back and forth to understand how a particular input affects system behavior. This effort leads to a loss of continuity when reading the specification.

		Ranking for Readability		Rank for Ease of Editing	
Background	Subjects	Macro	Flat	Macro	Flat
Aeronautics	1	1	2	1	2
	2	2	1	1	2
	3	1	2	1	2
	4	2	1	1	2
	5	2	1	1	2
	6	2	1	1	2
Computer Science	7	1	2	1	2
	8	2	1	1	2
	9	2	1	2	1
	10	2	1	1	2
	11	1	2	1	2
	12	1	2	1	2

Table 2. Subjective Rankings for Macros Experiment

This ambiguity regarding the usefulness of macros was further reflected in the subjects' rankings, shown in table 2. Roughly half (5) of the subjects found that the specification with macros was more readable than the corresponding flat specification. We had anticipated that most of the computer science students would prefer to read the macro specification, due to their presumed familiarity with the macro mechanism in programming languages. However, this 50/50 split found among the computer scientists as well, with only 3 of 6 computer scientists preferring the macro specification for readability. With respect to editing, however, 11 of the 12 subjects found that it was

easier to edit a specification using macros. One point that should be stressed is that macros can potentially become more useful when dealing with large systems, much larger than those specified in this experiment. Reading large specifications may have a significant impact on subjects' preferences.

Based on our results, we hypothesize that reaction to the use of macros would in fact be different if specifications were reviewed using an automated tool. The most common complaint recorded regarding the readability of macros was navigation difficulty, which could be assisted perhaps by hyperlinks in a tool. A tool could also automatically expand a macro in place, eliminating the need to look at several macros when reading a specification. The benefits of macros, particularly in writing a specification, are unmistaken. Tool support can help ameliorate their drawbacks.

4.4 Events

Subjects generally did not perform as well on the events experiment, which was expected. A histogram of subject performance is shown in figure 31. As far as general trends, half of the subjects either skipped question 6 or answered it incorrectly. Question 6 restricted the subjects to the eventless specification, and asked them to simulate the state machine model in a given environment. Ambiguity about the semantics of the eventless specification may have been a factor here. Interestingly, four of the six computer science students answered a comparable question of the events specification (question 11) incorrectly, while every aerospace engineer answered this question correctly. Based on a two-tail t-test, we can show at the 97% confidence level that there is a statistically significant difference between the performance of computer science and aeronautics students on question 11. Perhaps then aerospace engineers find that the use of events creates more readable specifications, whereas computer scientists, who had difficulty with both questions 6 and 11, simply have problems simulating several state machines that operate concurrently, with or without internal events. However, this theory not only seems unlikely, but is almost impossible to rationalize given the small number of subjects sampled.

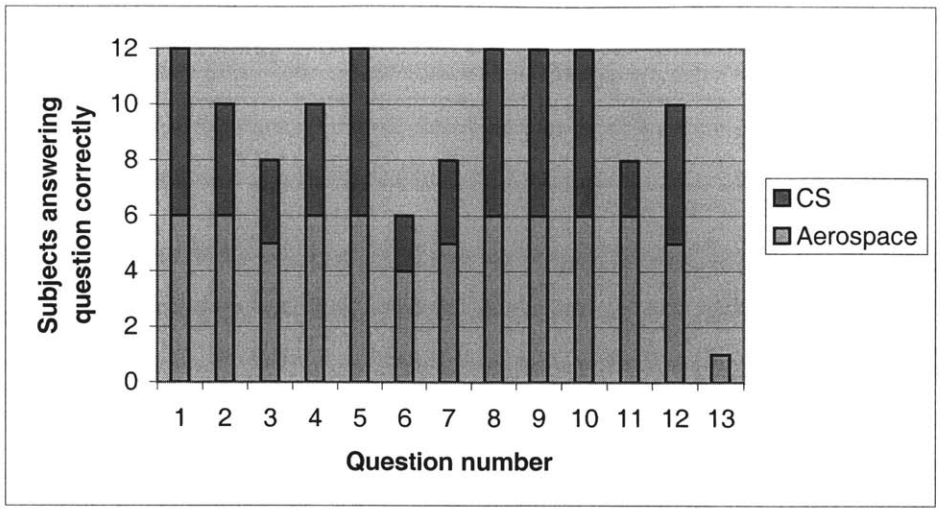


Figure 31. Histogram of Subject Performance for Events Experiment.

One interesting observation of the objective questions is that eleven subjects incorrectly answered question 13, which asked them to add a transition to the events specification. Most of the errors arose from the fact that subjects did not correctly see that the new transition required an internal event in the trigger, and instead used only a trigger condition. Question 13 specifically required subjects to change the specification such that the Alarm state machine would transition to *On* when the Altitude state machine is in the *Hazard* state. Figure 32 shows a correct response to the task, as well as a common response given by subjects. Because the transition depends on another state machine (i.e. Altitude) in the model, an internal event is necessary to communicate between the state machines. Most subjects supplied only a trigger condition, without a trigger event. This type of response is appropriate in an eventless specification, however. As a matter of note, every subject responded correctly to a comparable task in the eventless specification (question 8). Nonetheless, it seems clear from the responses that the subjects were unclear about the use of internal events. This observation is consistent with Leveson's experience with TCAS where they noted the difficulty they had in reading and writing specifications using internal events.

State	Trigger (Event)	(Condition)	Output Event
On	ALTITUDE HAZARD		
Off	ALTITUDE UNKNOWN, ALTITUDE ABOVE, ALTITUDE BELOW		

Correct Response

State	Trigger (Event)	(Condition)	Output Event
On		Altitude = Hazard	
Off		Altitude ≠Hazard	

Common Subject Response

Figure 32. Responses to Question 13 of Events Experiment.

For the first three questions, subjects were given access to both the events and the eventless specification. Eleven of the twelve subjects chose to use either the events specification or both specifications. Only one subject chose to use only the eventless specification. It appears as though subjects may feel comfortable using a specification with events, whether or not they have an adequate understanding of its semantics. In future work, it would be interesting to see whether subjects choose to use a specification using events when answering questions about a large system.

When asked to identify advantages of using internal events, most subjects felt that internal events made it easier to see how state machines affect each other. Internal events explicitly describe interactions in the state machine model, which can make a specification more readable. Defining these interactions without events can be difficult.

However, subjects did not appreciate the increased size associated with the events specification. Using internal events can lead to repetitive specifications, which comes at the expense of concision and readability. The triggers themselves are also more complex

than their eventless counterparts, in that the triggers have two parts, an event and a condition.

The greatest problem that subjects had with using events was determining when it is necessary to generate an output event (this, of course, was not in issue in the eventless specification). This difficulty was very apparent in the modification exercise, as described earlier. It seems that more training than that given in this experiment is necessary before one is able to use internal events effectively.

		Ranking for Readability		Ranking for Ease of Editing	
Background	Subject	Events	Eventless	Events	Eventless
Aeronautics	1	2	1	2	1
	2	1	2	2	1
	3	1	2	2	1
	4	2	1	2	1
	5	1	2	2	1
	6	1	2	1	1
Computer Science	7	2	1	2	1
	8	1	1	2	1
	9	1	2	2	1
	10	2	1	2	1
	11	2	1	2	1
	12	1	2	1	2

Table 3. Subjective Rankings for Events Experiment.

The results discussed thus far made it difficult to conclude anything about the readability of events, and this difficulty continued after reviewing the subjective rankings in this experiment, shown in table 3. Roughly half of the subjects (7/12) felt that the specification with events was more readable, the other half (5/12) preferring to read the eventless specification. This lack of preference was consistent across subject background, which was actually surprising, as it does not match our previous experiences with using internal events to create complex specifications. Subject preferences regarding events may need to be investigated in greater detail in the future.

Given these observations, we can hypothesize why subjects may prefer to read a specification with internal events. While potentially error-prone, internal events provide more salient information regarding how a state machine is to be executed. This type of information is not as explicit in an eventless language, so the reader may at first find a specification that uses events to be more readable.

Despite any ambiguity regarding the readability of internal events, ten of the twelve subjects found it easier to edit the eventless specification, which is consistent with the objective portion of our results discussed earlier.

These seemingly conflicting results about the readability of events made it difficult to reject the null hypothesis, which would imply that events do in fact affect readability. However, these results may just be random, and so we cannot accept the null hypothesis either – events may be (and likely are) contributing factors to a language's readability.

4.5 Hierarchies

Subjects again performed quite well in this part of the experiment, as shown in figure 33. The only noticeable deviation is in question 2, where half of the subjects (mostly computer scientists) responded incorrectly. The question asked subjects to identify those states that can transition directly into state *Time*. This task is fairly straightforward using the flat specification, as every transition is explicitly defined. However, as shown in figure 34, the transition from *Dead* to *Time* is not explicitly defined. The state machine transitions from *Dead* to *Watch On*, and the default state of *Watch On* is Time. Those subjects that answered this question incorrectly used the hierarchical specification, and failed to see that *Dead* can transition into *Time*.

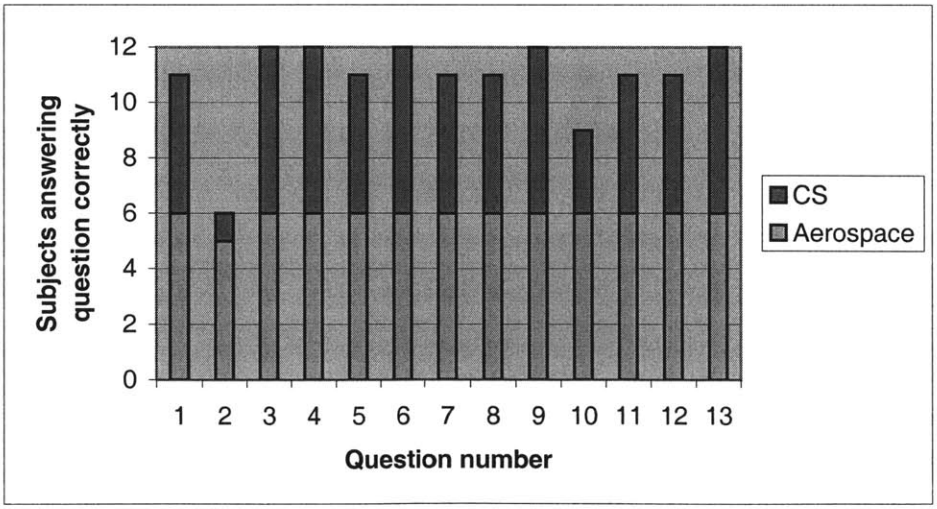


Figure 33. Histogram of Subject Performance for Hierarchy Experiment.

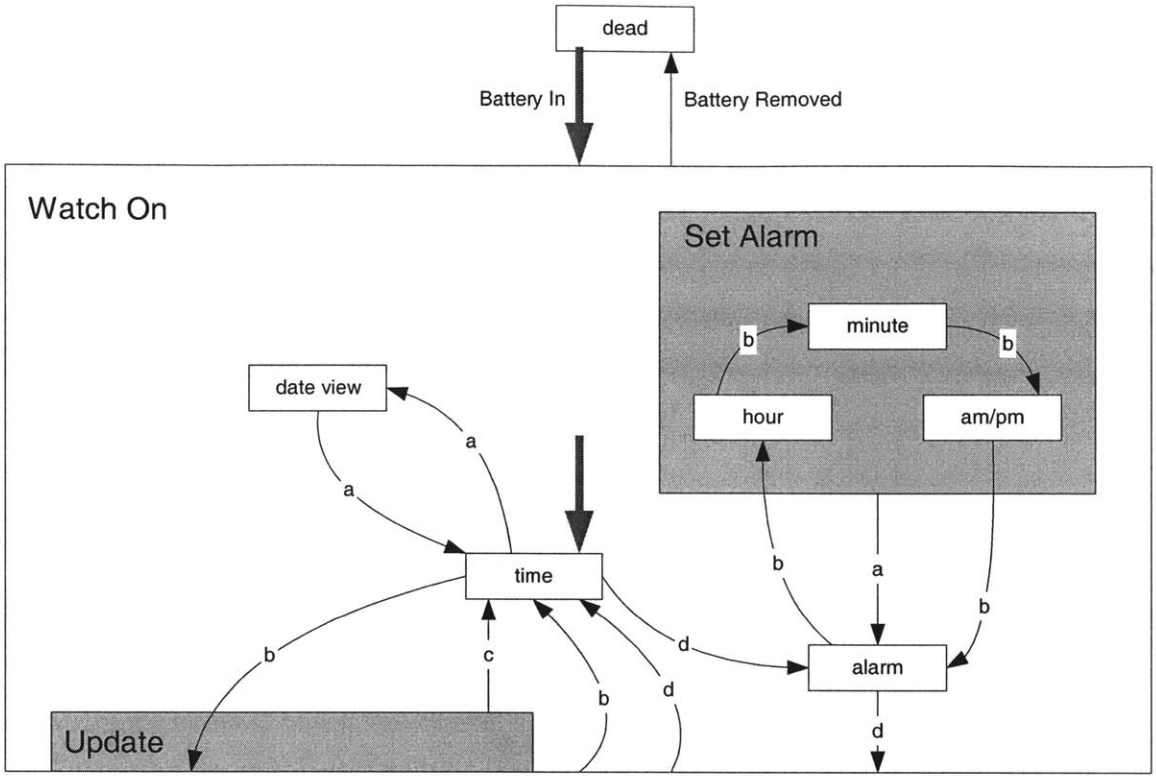


Figure 34. Transition from *dead* to *time* in hierarchical specification.

Subjects were given access to all specifications for the first three questions. Three subjects chose to use only the hierarchical, while the others used both. There did not seem to be an absolute preference for either specification.

Subjective feedback about the experiment was remarkably consistent. Every subject found that the flat specification was difficult to use due to the number of transitions in the state machine. This observation was made with respect to the graphical representation used, but may simply have to do with the fact that the state machine, regardless of its representation, possesses too many explicit transitions. Furthermore, only two of the subjects felt that special attention had to be paid when using the hierarchical specification in order to notice every transition in the state machine. The rest found no problems with the hierarchical state machine, which is surprising given the number of people that answered question 2 incorrectly, as described above.

As far as subjective rankings, every subject found the hierarchical specification to be more readable than the flat specification, and all subjects except three found it to be

easier to edit as well. These rankings are a clear indication that subjects were more comfortable using the hierarchical specification. However, these rankings are interesting given that so many subjects unknowingly made errors reading the hierarchical specification. When asked, all twelve subjects did not believe that flat specifications are appropriate for complex systems, which is easy to rationalize. Clearly any scalable state-based specification language must be hierarchical. However, it appears as though the behavior of the state machine should be more explicit in hierarchical specifications.

4.6 Perspective

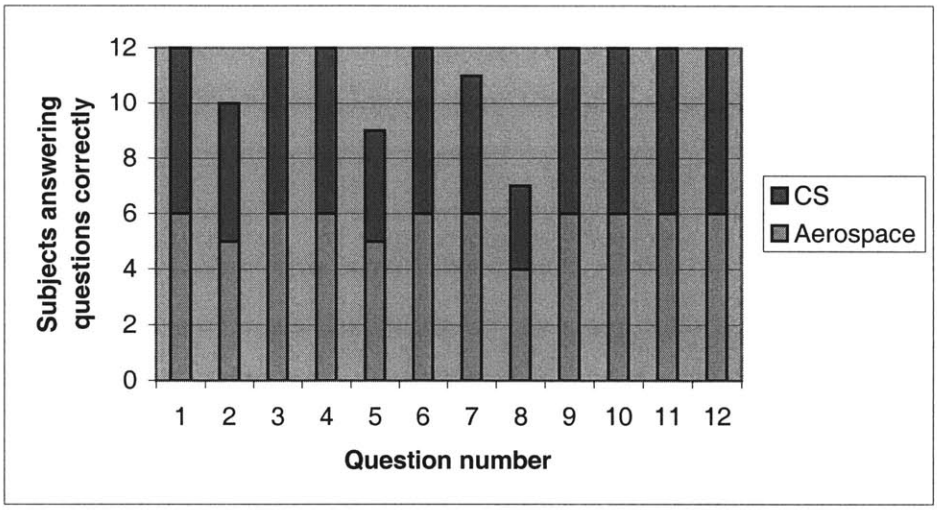


Figure 35. Histogram of Subject Performance for Perspective Experiment.

As shown in figure 35, subjects generally performed well on this experiment. The only question that drew a noticeable number of incorrect responses was Question 8, which restricted subjects to the Coming-from specification. The question asked subjects to determine whether it was possible for the state machine to be in a certain state, given a certain environment, and five subjects failed to produce a correct answer. There was no correlation between subject background and those that answered the question incorrectly. This observation provides some indication that subjects may have greater difficulty using a coming-from perspective. In fact, a total of 9 errors were made by subjects answering the questions for which they were restricted to the coming-from specification (questions 5-8), whereas every subject answered every question correctly for which they were

restricted to the going-to specification (questions 9-12). These results suggest that the going-to specification is more readable.

We were particularly interested in seeing which specification(s) subjects chose to use to answer the first four questions, for which they were given access to both specifications. All subjects used both specifications at some point, which is not surprising given that some questions are obviously better suited for one of the two specifications. For example, if asked which states can transition directly to Wait Mode, it is much easier to respond using the Coming-From perspective. However, for those questions which required the same amount of effort with both specification (e.g. "when will the system transition from state A to B?"), eight of the twelve subjects chose to use the going-to specification. Engineers may be a little more comfortable reading a state machine description with a going-to perspective, though the evidence found here is not everwhelming.

However, when asked whether one perspective was more intuitive than another, only five subjects responded that the going-to perspective provided a slightly more intuitive way to describe state machine behavior – the rest felt that the two views were basically interchangeable and that one was not more readable than the other. Of course, most subjects commented that certain types of questions were better addressed by one specification, which is an obvious observation.

When asked whether it is worthwhile to provide both perspectives in a specification, only three subjects felt that both views would be helpful. Most subjects felt that it would be better to become accustomed to one perspective rather than to switch between two. Based on the results of this experiment, it seems that the going-to would be the more readable perspective to provide. However, it may be the case that a going-to perspective is easier to use when answering questions about the details of a system's behavior – these are the types of tasks that were tested in this experiment. It has been our experience that when trying to develop a general understanding for a system's behavior, a coming-from perspective is easier to use.

Based on these experiments, we hypothesize that subject reaction to the use of different perspectives would be different in an automated environment. Certainly some

questions are easier to answer when considering a specification with one perspective rather than another, but subjects noted an aversion to using two specifications to answer questions. In an automated environment, these perspectives can be viewed without hassle, allowing the analyst to use whichever perspective is more appropriate without sacrificing convenience.

4.7 Experimental Error

As a human experiment, there are of course several potential sources of experimental error. Most of these have been discussed in the previous two chapters, but the most significant will be summarized here. We executed the experiment in spite of these factors, as we were interested in a practical experiment – one where subjects are placed in realistic scenarios and asked to perform realistic tasks. We recognize that this comes at the expense of a perfectly controlled experiment.

Familiarity with the subject material is certainly a source of error. Subjects, particularly the aerospace students, may be familiar with the concepts and terminology used in the specifications which may affect how readable they found the specifications. For example, a subject may have seen specifications for systems similar to those used in this experiment, and may then be biased towards a notation similar to that seen before. More generally, subjects may have experience with notations similar to those tested in this experiment, which would certainly affect their performance in the experiment.

Subject familiarity with state machines, and in particular internal events, likely affected our results. By providing an introduction to state machines, we hoped to bring subject knowledge to a common level. However, those more familiar with state machine terminology may have different preferences for notations. We attempted to analyze the effect of subject background on notational preferences, though no significant observations were made. Perhaps testing a larger subject pool would bring to light any correlation between background and performance in this experiment.

We attempted to control the experimentation conditions. All experiments began in the same office. However, due to distractions, three subjects were moved to neighboring offices. This adjustment should not have a significant impact on our results, but it is nonetheless noted here.

Perhaps the biggest source of experimental error is in the specifications tested themselves. Various studies have shown how factors such as modularization, naming conventions, fonts, etc. can affect readability. We decided these factors based on what we felt would be the most readable, which is of course subjective. With particular attention to the macro experiment, the amount of modularization used may have a dramatic impact on the specification's readability, as established in previous work. We would have liked to test the affect of various amounts of modularization on specification readability, but again sacrificed this for duration considerations.

As discussed previously, the length of the experiment was between 3-4 hours depending on the subject. This duration certainly affected the subjects' performances, and potentially their subjective rankings of the specification's readability. We encouraged the subjects to take a break after each part of the experiment to lessen any effects of boredom, but it was likely still an influential factor.

The difficulty of the questions asked is another source of experimental error. The questions, though involving realistic tasks, may have been too easy to test the readability of specifications in difficult scenarios. Most subjects performed very well on the objective portion of the experiment, which made it difficult to identify types of common errors, or to establish a correlation between subject performance and subject background. Furthermore, we recognize that we do not address all aspects of readability in our question set, as this is a difficult if not impossible task.

A final source of experimental error lies in the size of the specifications used. Whether or not a feature is readable may depend entirely on whether it is used in large or small specifications. Again, we are concerned with the readability of large system specifications, but in an effort to reduce the length of the experiment, we were forced to test only small specifications.

4.8 Recommendations

Based in part on these sources of experimental error, we have developed several recommendations for repeating this experiment. First, it would be worthwhile to test subjects with a non-technical background. Often times specifications for software

systems are written not by computer experts, but rather by non-experts. The readability of formal languages to non-experts may therefore be important if formal methods are adopted. On a related note, testing a larger, more statistically significant number of subjects would also be helpful in drawing conclusions from the experiment. A larger subject base might also make relationships between a subject's background and his/her preferences more apparent, if such relationships exist.

Providing a more thorough training of subjects in state machine behavior should also improve the quality of this experiment. The brief introduction provided here appeared adequate for our subjects, though we recognize that these subjects may be more well-versed in the use of state machines than an average engineer, and certainly more so than specification developers without a technical background. Furthermore, industries adopting formal specification languages would likely provide a more substantial background than provided here, and this may influence a subject's notational preferences.

Another recommendation for repeating this experiment is to better enforce the rules of the experiment. As described earlier, some subjects "cheated," using specifications other than that which they were restricted to using because they found it unreadable. Though this did provide interesting results for our experiment, subjects should not be permitted to use specifications other than those prescribed in the experiment design.

It may be worthwhile to include a time limit for the experiment. Though we had several reasons for eliminating a time limit, such a constraint may help to alleviate the effects of tedium in the experiment. A time limit would also make it possible to run the experiment with several subjects concurrently. Running the experiment in such a manner would help control environment variability in our results.

A final recommendation for repeating this experiment is to use less discipline-related experimental material. We focused our experiment material specifically on the aerospace field. However, by testing systems with a broader appeal, experimenters may be able to make more general conclusions about the readability of the various notations tested here.

5. CONCLUSION

One of the major obstacles facing the adoption of formal methods by the industrial community is readability. Most formal specifications are simply difficult to read, which limits their use as standalone specifications. We have attempted to alleviate this problem by identifying those features of state-based specification languages that most affect readability. We have surveyed different means of describing the various parts of a state machine model, and tested the readability of each in a human experiment. To date, claims about a language's readability rely on subjective or theoretical evidence, without any sort of empirical evidence. Our work has provided a first step in providing empirical evidence about readability, which we feel is essential in evaluating the usefulness of state-based languages. Readability is a complex property, which we do not feel can be predicted theoretically, but rather is best evaluated by human experience.

After surveying five state-based specification languages, we identified several distinguishing features, and ultimately decided to test how the following affect readability: representation of the state machine layout, expression of trigger conditions, macros, internal events, hierarchical abstractions, and perspective. A brief summary of our results follows:

> **Representation**: We tested tabular, graphical, and textual representations of state machines. Eleven of the 12 subjects found either the graphical or the tabular representation to be the most readable, with only 1 subject preferring to read the textual specification. This observation may imply that textual specifications, though popular in industry, are not conducive to readability. Subjects were given their choice of specifications to answer two questions – every subject chose to use the tabular specification for the first question, and 10 of the 12 chose the table for the second question.
>
> **Conditions**: We tested textual, graphical, tabular, and logical expressions of trigger conditions. Nine of the 12 subjects found the tabular specification to be the most readable, and 7 of the 12 ranked the logical specification as the least readable. Furthermore, 10 of the 12 subject felt that it is worthwhile to provide an analyst with two different representations of trigger conditions, and that one of the two should be tabular. The two subjects that preferred the simplicity of a single representation said that this representation should be tabular as well. Our results imply that tables are the most conducive to the readability of trigger conditions.

Macros: Results at first seemed to imply that macros were not conducive to readability. Subjects were given their choice of specifications to answer two questions, and 11 of the 12 subjects chose to use the flat specification. Furthermore, only 5 of the 12 subjects ranked the macros specification as the more readable. However, 11 subjects ranked the macros specification as the easier of the two to edit. Based on these results and subject interviews, it appears that macros are useful when writing a specification, but that they can become difficult to navigate when reading a large specification. Prohibiting nested macros may be a compromise for this situation. However, using macros in an automated environment may alleviate much of the difficulty encountered using macros in this experiment. Navigation could be simplified or even unnecessary when specifications are reviewed with an automated tool.

Events: Results regarding the readability of internal events were inconclusive. Seven of the twelve subjects ranked the events specification as more readable, whereas ten of the twelve found it easier to modify the eventless specification. Eleven subjects were unable to edit the events specification correctly, which implies that subjects were not clear about how to execute internal events. More training in the use of internal events than that provided here may be necessary.

Hierarchies: Interestingly, all 12 subjects agreed that a hierarchical specification is easier to read than a flat specification, and that hierarchical abstractions are absolutely necessary to specify complex systems. However, half of the subjects unknowingly made errors reading the hierarchical specifications, errors that were not made reading the flat specification. These results imply that notations describing state machine hierarchies may need to convey information in a more salient manner.

Perspective: Seven subjects felt that the going-to and coming-from perspectives were interchangeable, and 9 subjects responded that it is more desirable to become accustomed to one perspective rather than be provided with both, despite the fact that certain questions were clearly easier to answer using one perspective over the other. However, our results suggest that subjects may actually prefer access to both perspectives in an automated environment.

Our subject pool consisted of graduate students in either aeronautics or computer science. We attempted to find a correlation between a subject's background and his/her performance in the experiment. For the most part, however, we were unable to do so. Though time was not officially recorded, computer science students finished the experiment in roughly 3 hours, while aerospace students finished in roughly 3.5-4 hours, and with similar accuracy. This observation may indicate that a background in state machines, which most aerospace students do not have, assists with reading state-based

state machines. This may also indicate that the tasks involved with reading and editing specification are not addressed in aerospace curricula.

The most frequently referenced weakness in student-based experiments is that the results are not generalizeable to industry. However, we do not feel that this weakness applies to this experiment. The amount of training in state machine behavior given to subjects in this experiment is comparable to training that might be given to professional software developers. It in unlikely that engineers in the computer science (aerospace) industry know any more about formal methods than the computer science (aerospace) students tested here. In fact, most proponents of formal methods argue that these specification languages can be used with a minimal amount of training, making it appropriate to generalize our results to the industrial community.

As an assessment of formal specification readability, these results are preliminary. We would ultimately like to run more extensive experimentation focusing on 2-3 of these features. Based on our results, we will likely focus on conditions, macros and internal events. Complex conditions seem to be more readable when expressed in a table, but we would like to test this with a statistically significant number of subjects before claiming it as a conclusion. The macros portion of our experiment was inconclusive regarding their effect on readability. Though they seem to be extremely useful when writing and editing specifications, there seems to be difficulty reading specifications with macros, which we would like to address in future work. Specifically, we would like to investigate how various levels of modularization (i.e. nested macros) affect a specification's readability. Macros also are one feature whose utility is dependent on the size of the specification. We would like to investigate how macros can affect the readability of large-scale system specifications. Future work will also likely address the use of internal events. Again, while most subjects found it easier to edit an eventless specification, feelings were mixed regarding whether it is easier to read an eventless specification. The TCAS experience found that reading large specifications was impeded by internal events. We would like to provide empirical evidence to this claim by extensively testing the readability of events in large specifications.

We are not planning to investigate the readability of hierarchies or perspective in future work. Our results, as well as previous experience, show that hierarchical abstractions are essential if formal methods are to scale, so we do not feel that any insight would come from further experimentation. We have learned, however, that notations describing these hierarchies should use simpler semantics when describing state machine behavior. Perspective does not appear to be a significant factor affecting readability, and so it does not seem worthwhile to investigate further.

Our experiment was well designed and executed, and provided useful results. However, there were lessons learned in this work that will be applied to future experimentation. Again, we will focus on 2-3 features, rather than the 6 investigated here. Investigating several features made it difficult to study any one in great detail, due to concerns about experiment duration. We will also test a more statistically significant number of subjects. This project was short-term, and as such it was difficult to test more than twelve subjects, as each experiment ran between 3-4 hours. A greater subject base is obviously important if any conclusions are going to be made about language readability. Testing more subjects may also bring to light a correlation between a subject's background and his/her notational preferences.

We tested for four different skills in our assessment of readability. Future experimentation will likely include several objective questions testing each skill. This will help us better analyze subjects' performance on

Duration of the experiment was a major concern throughout the experiment design, and will be addressed in future experimentation as well. Specifically, we would like to better pretest the experiment so that we can be assured of its duration. Another possible solution is to use time limits as a measure of readability. For example, we can require that subjects spend no longer than 90 seconds on a question, or that they spend no longer than 20 minutes on a section.

For the most part, the amount of training provided to subjects was sufficient. However, there still seemed to be some confusion about the operation of internal events. Future experimentation will better address this aspect of training, so that we can more accurately determine how the events impact readability. It is unfair to conclude that

internal events are difficult to use if subjects are not adequately instructed in their behavior. This need for more exhaustive training in using events implies that they are harder to use than other mechanisms. However, it is important to note that the amount of training provided here was minimal compared to the amount of training that would likely be tolerated by industry. When understood, internal events may actually enhance readability.

Of course, the most important issue that will be addressed in future work is the testing of large scale system specifications. Scalability is one of the largest obstacles facing the adoption of formal methods, and was not adequately addressed in this experiment. For example, we tested the readability of a flat specification. While some subjects found a flat specification easy to read when describing a state machine with ten states, it is clearly inappropriate when describing a state machine with 10^{30} states.

After determining which notations and features are conducive to making large system specifications readable, we can, in the long term, incorporate these conclusions into the design of a more ideal state-based specification language. Such a language will be able to produce specifications that not only possess the analytical power of a state machine model, but that can function well as standalone specifications. These characteristics will encourage the adoption of formal methods by aerospace and other industries, which can ultimately improve the reliability and safety of modern software systems.

Bibliography

[1] Bingham, J. and Davies, G. A Handbook of Systems Analysis. Macmillan Publishers Ltd., London, 1972.

[2] Bowen, Jonathon and Hinchey, Michael. *Ten Commandments of Formal Methods.* IEEE Computer. vol 28, no 4, April, 1995. pp. 56-63.

[3] Brooks, F. *The Computer Scientist as Toolsmith: Studies in Interactive Computer Graphics.* Proceedings of the International Federation of Information Processing Congress '77, Toronto, Canada, August 1977. pp. 625-634.

[4] Brooks, Ruven. *Studying Programmer Behavior Experimentally: The Problems of Proper Methodology.* Communications of the ACM. vol 23, no. 4, April 1980. pp. 207-14.

[5] Clarke, Edmund and Wing, Jeannette. *Formal Methods: State of the Art and Future Directions.* ACM Computing Surveys. vol 28, no. 4, December 1996. pp. 626-43.

[6] Fenton, N. and Kaposi, A. *An Engineering Theory of Structure and Measurment.* Software Metrics – Measurement for Software Control and Assurance. Elsevier Science, London, 1989. pp. 27-62.

[7] Finney, K., Rennolls, Keith, and Fedorec, Alex. *Measuring the Comprehensibility of Z Specifications.* The Journal of Systems and Software 42. Elsevier Science Inc., 1998. pp. 3-15.

[8] Finney, K., Fenton, N., and Fedorec, A. *Effects of Structure on the Comprehensibility of Formal Specifications.* IEE Proceedings - Software. vol 146, no. 4, August 1999. pp. 193-202.

[9] Fitter, M. and Green, T.R. *When Do Diagrams Make Good Computer Languages?* International Journal of Man-Machine Studies. vol. 11, 1979. pp. 235-261.

[10] Green, T.R. *Design and Use of Programming Languages.* NATO ASI Series. vol F22. Sofware System Design Methods. Springer-Verlag, Berlin. 1986. pp.224-38.

[11] Harel, David. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming 8. Elsevier Science Publishers B.V., North Holland. 1987. pp. 231-274.

[12] Harel, David et al. *Statemate: A Working Environment for the Development of Complex Reactive Systems.* IEEE Transactions on Software Engineering. vol 16, no. 4 April, 1990. pp. 403-14.

[13] Heimdahl, Mats and Leveson, Nancy. *Completeness and Consistency in Hierarchical State-Based Requirements.* IEEE Transactions on Software Engineering. vol 22, no. 6 June, 1996. pp. 363-77.

[14] Heimdahl, Mats, Leveson, Nancy, and Reese, Jon. *Experiences From Specifying the TCAS II Requirements Using RSML.* Proceedings of the 17th Digital Avionics Systems Conference, November 1998.

[15] Heitmeyer, Constance, Jeffords, Ralph, and Labaw, Bruce. *Automated Consistency Checking of Requirements Specifications.* ACM Transactions on Software Engineering and Methodology, vol 5, no. 3, July, 1996. pp. 231-261.

[16] Howard, Jeffrey. *Principles for Design of Software Engineering Visualization Tools.* Massachusetts Institute of Technology, S.M. Thesis, September 2000.

[17] Kitchenham, Barbara, Pfleeger, Shari, and Fenton, Norman. *Towards a Framework for Software Measurement Validation.* IEEE Transactions on Software Engineering. vol 21, no. 12, December 1995. pp. 929-944.

[18] Leveson, N., Heimdahl, M., Hildreth, H., and Reese, J. *Requirements Specification for Process Control Systems.* IEEE Transactions on Software Engineering. vol 20, no. 9, September 1994. (TCAS)

[19] Leveson, Nancy. Safeware: System Safety and Computers. Addison-Wesley Publishing Company, U.S. 1995.

[20] Leveson, Nancy. *SpecTRM: A CAD System for Digital Automation.* Proceedings of the 17th Digital Avionics Systems Conference. November, 1998. pp. B52-1 – 8.

[21] Leveson, N., Heimdahl, M., and Reese, J. *Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future.* SIGSOFT Foundations of Software Engineering '99, Toulouse, France. September, 1999.

[22] Leveson, Nancy. *Completeness in Formal Specification Language Design for Process Control Systems.* Proceeedings of Formal Methods in Software Practice Conference, August 2000.

[23] Leveson, Nancy. *Intent Specifications: An Approach to Building Human-Centered Specifications.* IEEE Transactions on Software Engineering. vol 26, no. 1, January, 2000. pp. 15-35.

[24] Leveson, Nancy. HETE System Specification. Massachusetts Institute of Technology, Course 16.358 (System and Software Safety) Class Handout. March 9, 2001.

[25] Lindsey, G. *Structure Charts: A Structured Alternative to Flowcharts.* SIGPLAN Notices. November, 1977. pp. 36-49.

[26] McIver, Linda and Conway, Damian. *Seven Deadly Sins of Introductory Programming Language Design.* Proceedings of the IEEE International Conference on Software Engineering: Education and Practice, 1996. pp.309 –316

[27] Schneiderman, Ben. *Measuring Computer Program Quality and Comprehension.* International Journal of Man-Machine Studies. vol 7, no. 4, 1977. pp. 465-78.

[28] Sherry, Lance, Youssefi, David, and Hynes, Charles. *A Formalism for the Specification of Operationally Embedded Reactive Avionic Systems.* HSR FD G&C Task 4 – VMS Structure Development. Honeywell, October, 1995.

[29] Sipser, Michael. Introduction to the Theory of Computation. PWS Publishing Company, U.S., 1997.

[30] Sowmya, Arcot and Ramesh, S. *Extending Statecharts with Temporal Logic.* IEEE Transactions on Software Engineering. vol 24, no.3 March, 1998. pp. 216-231

[31] Tufte, Edward. Envisioning Information. Graphics Press, CT. 1990.

[32] Tenny, T. *Program Readability: Procedures versus Comments.* IEEE Transactions on Software Engineering. vol 14, no. 9, September 1998. pp. 1271-9.

[33] Vinter, Rick, Loomes, Martin, and Kornbrot, Diana. *Applying Software Metrics to Formal Specifications: A Cognitive Approach.* Proceedings of the 5th International Software Metrics Symposium. 1998. pp. 216-223.

[34] Wright, Patricia. *Feeding the Information Eaters: Suggestions for Integrating Pure and Applied Research on Language Comprehension.* Instructional Science, an International Journal. vol 7, no. 3, 1978. pp. 249-312.

[35] Zimmerman, Marc, Rodriguez, Mario, Ingram, Benjamin, Katahira, Masafumi, de Villepin, Maxime, and Leveson, Nancy. *Making Formal Methods Practical.* Proceedings of the 19th Digitial Avionics Systems Conference. October, 2000.

Appendix A: Experiment Questions

Subject Number:

Background Assessment

1. Describe any experience (research and/or class work) you have with:

   Discrete math and/or logic

   Formal methods

   Complexity theory

   Control Systems

   Requirements or Software specification languages

   Programming

2. What was your primary course of study at your undergraduate institution?

3. What is your primary course of study in graduate school?

Representations

As a reminder, if you find that it is taking longer than 1 minute to answer a question, please skip it and move on to the next one.

(Any specification)

1. Which transitions can be affected by the Wheel Spin Rate input? Give source and destination of transitions. (2)

- ❑ Table
- ❑ Text
- ❑ Graph

2. Unfortunately there is an error in this specification. Two transitions out of Acquire mode can be satisfied simultaneously. Which transitions are they (give destination states of the transitions)?(3)

- ❑ Table
- ❑ Text
- ❑ Graph

(Textual specification only)

3. Which part of the specification details the conditions under which the system should transition from Orbit Night Mode to Spinup Mode (label lines in the specification)? (1)

4. If we are currently in Spinup Mode, what will cause a transition to Reorient Mode?(2)

5. Add the transition from Wait Mode to Orbit Day Mode if the system is in Eclipse, Time(In Mode) >= Wait Mode Delay, and the Optical System is tracking.(4)

6. With the addition of the previous transition, two transitions out of Wait Mode can be satisfied simultaneously. Give any change that can be made to the specification to ensure that only one transition out of Wait Mode is satisfied at any given time?(4)

(Graphical specification only)

7. Which states can transition directly to Ground Control Mode?(2)

8. If I am in Orbit Night Mode, which of the following inputs are not relevant to any immediate transition?(2)

Optical System Tracking/Not Tracking,
Momentum Error,
Sine Sun Elevation
Tumble Rate

9. Add the transition from Wait Mode to Orbit Day Mode if the system is in Eclipse, Time(In Mode) >= Wait Mode Delay, and the Optical System is tracking.(4)

10. With the addition of the previous transition, two transitions out of Wait Mode can be satisfied simultaneously. Give any change that can be made to the specification to ensure that only one transition out of Wait Mode is satisfied at any given time?(4)

(Tabular specification only)

11. Which states can transition directly to Reorient Mode?(2)

12. If the system is in Orbit Day Mode, what transition would take place under the following conditions?(3)

The system is not in an Eclipse
The Optical System is tracking
The Paddles are Deployed.
The Time in Orbit Day Mode > Orbit Day Mode Delay
The Sine Sun Elevation > Coarse Sun Elevation Error
The Sine Sun Azimuth = Coarse Sun Azimuth Error

13. Add the transition from Wait Mode to Orbit Day Mode if the system is in Eclipse, Time(In Mode) >= Wait Mode Delay, and the Optical System is tracking.(4)

14. With the addition of the previous transition, two transitions out of Wait Mode can be satisfied simultaneously. Give any change that can be made to the specification to ensure that these two transitions cannot be satisfied at the same time.(4)

Representation Evaluation

1. Rank the textual, graphical, and tabular representation in terms of readability.

2. Were certain forms good for certain tasks, or did you consider one model to be the single most useful?

Conditions

As a reminder, if you find that it is taking longer than 1 minute to answer a question, please skip it and move on to the next one.

(Any specification)

1. Describe two scenarios that would cause the Climb FMS Speed Mode to be Edit. (label columns in the table) (2)

- ❑ Table
- ❑ Gate
- ❑ Text
- ❑ Logic

2. Could the Climb FMS Speed Mode be Max Climb under the following conditions? (3)
   Engine Out is Engaged
   Requested Climb FMS Speed Mode = Economy
   Flight Phase transitions from Takeoff to Climb

- ❑ Table
- ❑ Gate
- ❑ Text
- ❑ Logic

(Tabular specification only)

3. Which part of the specification specifies that the Climb FMS Speed Mode will be the Default when the flight phase transitions from Climb to Descent (label rows and columns in table)?(1)

4. If the FCC Engaged Mode is Altitude Hold Speed, what additional conditions are necessary in order for the Climb FMS Speed Mode to be Default? (2)

5. Could the Climb FMS Speed Mode be Default under the following conditions? (3)
   FMS Mode is Lateral Only
   Engine Out is Not Engaged
   Flight Phase is Cruise
   Economy is requested for the FCC Speed Mode

6. Suppose that in order for the Climb FMS Speed Mode to be Edit, the FMS Mode must be Lateral-Vertical (this is in addition to the existing requirements). What changes should be made to the specification to reflect this behavior? (4)

(Textual specification only)

7. Which part of the specification specifies that the Climb FMS Speed Mode will be the Max Climb when the Max Climb is requested (label lines in the specification)? (1)

8. If the flight phase is Preflight, under what conditions will the Climb FMS Speed Mode be Economy? (2)

9. Under the following conditions,

FMS Mode is Lateral-Vertical
Engine Out transitions from Not Engaged to Engaged
Flight Phase is Cruise
Economy is requested for the Climb Speed Mode

could the Climb FMS Speed Mode be Default? Economy? (3)

10. Suppose we want to add a new mode for the Climb FMS Speed Mode, called Flex. The Climb FMS Speed Mode will be Flex if the FMS Mode is Lateral-Vertical, and the flight phase is either Takeoff, Climb, or Cruise. What additions should be made to the specification to reflect this behavior? (4)

(AND/OR gate specification only)

11. Which part of the specification specifies that the Climb FMS Speed Mode will be Default when the FMS Mode is Lateral Only (label inputs/gates in the specification)? (1)

12. If the flight phase transitions from Approach to Done, what additional conditions are necessary in order for the Climb FMS Speed Mode to be Default? (2)

13. Could the Climb FMS Speed Mode be Edit under the following conditions? (3)
there is no requested Climb FMS Speed mode
Flight Phase is Preflight
Economy is requested for the FCC Speed Mode

14. Suppose that in order for the Climb FMS Speed Mode to be Edit, the FMS Mode must be Lateral-Vertical (this is in addition to the existing requirements). What additions should be made to the specification to reflect this behavior? (4)

(Propositional logic specification only)

15. Which part of the specification specifies that the Climb FMS Speed Mode will be Economy when the Requested FCC Speed Mode is Economy and the Flight Phase is Preflight (label lines in the specification)?(1)

16. If Edit CAS is requested for the FCC Speed Mode, what must the flight phase be in order for the Climb FMS Speed Mode to be Edit? (2)

17. Could the Climb FMS Speed Mode be Economy under the following conditions? (3)
    Requested for the FCC Speed Mode is AFS Speed
    FCC Engaged Mode is Altitude Hold Economy Thrust
    Flight Phase transitions from Descent to Cruise

18. Suppose we want to add a new mode for the Climb FMS Speed Mode, called Endurance. The Climb FMS Speed Mode will be Endurance if the FMS Mode is Lateral-Vertical, there is no requested FCC Speed Mode, and the flight phase is either Takeoff, Climb, or Cruise. What additions should be made to the specification to reflect this behavior? (4)

Condition Evaluation

1. Rank the textual, tabular, graphical, and logical specification in terms of readability.

2. Do your preferences change with respect to ease of editing?

3. Did you find that certain forms were good for certain tasks, or were there specifications that you consistently found to be easy to use in the experiment?

4. Do you consider it worthwhile to have several representations available? If so, which ones?

Macros

As a reminder, if you find that it is taking longer than 1 minute to answer a question, please skip it and move on to the next one.

(Any specification)

1. What must the Vertical Guidance Type be if the Vertical Guidance Reference Altitude is to be the *Engine Out Driftdown Deceleration Altitude*? (1)

- ❑ Flat
- ❑ Macros

2. If the Final Approach prompt has been selected, is it possible for the Vertical Guidance Reference Altitude to be the *Altitude Constraint at the Destination*? (2)

- ❑ Flat
- ❑ Macros

(Flat specification only)

3. Describe one scenario that can cause the Vertical Guidance Reference Altitude to be the *Cruise Conflict Altitude*. (2)

4. Where is it stated that the Vertical Guidance Reference Altitude will be the *Clearance Altitude* if the Vertical Guidance Type is Airmass AFS and the Flight Phase is Takeoff (label rows and columns in the specification)? (1)

5. Can the Vertical Guidance Reference Altitude be Engine Out Driftdown Deceleration Altitude under the following conditions? (3)

Altitude Rate is 350 ft/sec
Engine Out is engaged
Vertical Guidance Type is Airmass PROF
Flight Phase is Descent
Aircraft Altitude is above Computed 2 Engine Max. Altitude

6. Suppose we want to add a conditions to the first table of the vertical guidance Vertical Guidance Reference Altitude specification (i.e. = *Climb Target Altitude*). The existing conditions will stand, but in addition, the FMS speed mode must either be in Edit mode or Economy mode. What additions must be made to the non-macro specification to accommodate this condition?(4)

(Macro specification only)

7. What defines a Climb Conflict Situation? (2)

8. Which part of the specification explains that the Vertical Guidance Reference Altitude will be the *Cruise Conflict Altitude* if: (1)

Flight phase is cruise
Clearance altitude < aircraft altitude – 250 ft.
Vertical guidance type is airmass PROF
FCC autopilot is engaged
Step climb is active, AND
Descent/approach capture/hold criteria are not satisfied

(label lines in the specification)

9. Under the following conditions,

Flight Phase is Descent
Final approach prompt has been selected
Non-precision VFR approach type has been selected
Aircraft Altitude is 24,000 ft.
2 Engine Maximum Altitude is 40,000 ft.
Clearance Altitude = 25,000 ft.
Descent Target Altitude = 25,000 ft.
Aircraft is below path approach level-off
Descent speed limit violation = FALSE
The Profile Descent variable is TRUE
FCC Autopilot is Engaged
Vertical Guidance Type is Airmass AFS

which of the following will be the value of the Vertical Guidance Reference Altitude? (Hint: Evaluate the *Descending* and *Descent Conflict Situation* macros first) (3)

a. Altitude Constraint at Destination
b. Below Path Approach Level Off Altitude

10. Suppose we want to add a condition to the fifth table of the vertical guidance Vertical Guidance Reference Altitude specification (i.e. = *Actual 2 Engine Maximum Altitude*). The existing conditions will stand, but in addition, the Thrust Limit must be FLX, Max, or Climb. Add this condition to the macro specification, by first creating a macro, and then referring to it in the table. (4)

Macros Evaluation

1. Which specification was easier to read? to edit?

2. Did you find that a specification was good for some tasks and not others, or did you find one specification consistently easier to use when answering questions in the experiment?

3. What are some advantages to using macros in specifications? disadvantages?

4. Considering the advantages/disadvantages of using macros, would you prefer specifications to include macros? Why?

Events

As a reminder, if you find that it is taking longer than 1 minute to answer a question, please skip it and move on to the next one.

State variables will appear in **bold** type.

(Any specification)

1. When will the system detect a fault in the **Controlled Device Status**? (1)

❑ Events
❑ Eventless

2. When **Digital Altimeter** transitions to *Invalid*, which other state variabes *can* be directly affected by this transition? (2)

❑ Events
❑ Eventless

3. When the Controls Reset occurs, which state variables are directly affected? (2)

❑ Events
❑ Eventless

(Eventless specification only)

4. When **Altitude** transitions to *Below Threshold*, which other state variables can be directly affected by this transition? (2)

5. Which part of the specification specifies that the **Altitude** cannot be determined if the **Analog Altimeter** and **Digital Altimeter** are invalid (label rows in table)?(1)

6. Assuming the following conditions:

> **Altitude** is in state *Below-Threshold*
> **Controlled Device Status** is in state *Off*
> **System Status** is in state *Operational*
> **Digital Altimeter** is in state *Invalid*
> Digital Altimeter Value = 3500 ft.

what state will the **Power Command Output** be in if Analog Status becomes invalid? (3)

7. Assume we want to add another possible value to the **Altitude** state variable called *Hazard*. **Altitude** will transition to *Hazard* if *either* of the altitude sensors (analog or digital) is Valid and records an altitude of less than 1000 ft. What changes need to be made to the eventless specification to reflect this addition?(4)

8. Assume now that we want to add a state variable called **Alarm**. **Alarm** has 2 states (*on, off*). The alarm will be *on* if the **Altitude** is in the *Hazard* state, and *off* at all other times. What additions need to be made to the eventless specification to reflect this?(4)

(Event specification only)

9. Which part of the specification explains that the **Altitude** is *Below Threshold* when Digital Altimeter is valid and measures an altitude of less than 2000 ft (label rows in table)? (1)

10. If the **Altitude Switch Status** is in state *Inhibited*, what additional condition will turn the **Watchdog Probe Output** on? (2)

11. What state will the **Power Command Output** be in after the following events are received? (3)

1. Startup
2. Device Signal = Off
3. Digital Status = Valid
4. Digital Altimeter Value = 2100
7. Analog Status = Invalid
8. Analog Altimeter Value = 2200
9. Device Signal = On

12. Assume we want to add another possible value to the **Altitude** state variable called *Hazard*. **Altitude** will transition to *Hazard* if *either* of the altitude sensors (analog or digital) is Valid and records an altitude of less than 1000 ft. What additions need to be made to the event specification to reflect this addition?(4)

13. Assume now that we want to add a state variable called **Alarm**. **Alarm** has 2 states, (*on, off*). The alarm will be *on* if the **Altitude** is in the *Hazard* state, and *off* at all other times. What changes need to be made to the event specification to reflect this?(4)

Event Evaluation

1. Which specification did you find easier to read? Why?

2. Which specification did you find easier to edit?

3. Do you think that your preferences would change when dealing with large specifications?

4. What are some strengths of the event specification? drawbacks?

5. What are some strengths of the eventless specification? drawbacks?

Hierarchies

As a reminder, if you find that it is taking longer than 1 minute to answer a question, please skip it and move on to the next one.

(Any specification)

1. Which states can transition directly into *alarm*?(1)

- ❑ Flat
- ❑ Hierarchical

2. Which states can transition directly into *time*?(1)

- ❑ Flat
- ❑ Hierarchical

3. If the watch is dead, and a battery is inserted, what state will the sytem transition to?(3)

- ❑ Flat
- ❑ Hierarchical

(Flat specification only)

4. Under what conditions will the system transition from (set alarm) *am/pm* to *alarm*?(2)

5. Suppose we want the watch to transition from *time* mode to (stop watch)-*on,lap* mode when the *c* button is pressed. What changes must be made to the flat specification to represent this behavior?(4)

6. If the stop watch is in state *off* and in *lap* mode, and button *b* is pressed, what state will the system transition to?(3)

7. If the stop watch is On, under what conditions will it be reset to *zero*? (2)

8. Suppose we want to divide the chime state into 2 different states – *visual chime* and *audio chime*. From the *alarm* state, the system will first transition into the *visual chime* state when the *a* button is pressed, and will then transition into the *audio chime* state when the *a* button is pressed again. If the *d* button is pressed while in the *visual* or *audio chime* states, the system should transition to the stop watch-*zero* state. What changes must be made to the flat state machine to reflect this behavior?(4)

(Hierarchical specification only)

9. Under what conditions will the stop watch transition from *regular* mode to *lap* mode?(2)

10. If I am updating the date setting of the watch (i.e. I am in the *date* state), under what condition will the system transition to the *time* state?(2)

11. If the watch is in the (stop watch) *zero* state, and the *d* button is pressed twice consecutively, what will be the final state of the system?(3)

12. Suppose we want the watch to transition from *time* mode to stop watch-run-*on* and to stop watch-display-*lap* mode when the *c* button is pressed. What changes must be made to the hierarchical specification to represent this behavior?(4)

13. Suppose we want to divide the chime state into 2 different states – *visual chime* and *audio chime*. From the *alarm* state, the system will first transition into the *visual chime* state when the *a* button is pressed, and will then transition into the *audio chime* state when the *a* button is pressed again. If the *d* button is pressed while in the *visual* or *audio chime* states, the system should transition to the stop watch-*zero* state. How could a superstate *chime*, composed of the *visual* and *audio chime* states, be included in the hierarchical state machine to reflect this behavior?(4)

Hierarchies Evaluation

1. Which specification was easier to read?

2. Which specification was easier to edit?

3. Would you prefer access to a flat specification, or a hierarchical specification? Why?

4. What are some difficulties you had with the hierarchical specification? the flat specification?

5. Do you think flat specifications are suitable for complex systems?

Perspectives

As a reminder, if you find that it is taking longer than 1 minute to answer a question, please skip it and move on to the next one.

(Any specification)

1. From which states can the system enter Wait Mode?(1)

❑ Coming From
❑ Going to

2. Under what condition(s) will the system transition from Acquire Mode to Reorient Mode?(2)

❑ Coming From
❑ Going to

3. When will the system transition from Orbit Night Mode to Spinup Mode?(2)

❑ Coming From
❑ Going to

4. If the system is in Orbit Day Mode, which states can it transition to directly?(1)

❑ Coming From
❑ Going to

(Coming-from specification only)

5. What conditions will trigger a self transition from Detumble Mode to Detumble Mode?(2)

6. From which states can the system enter Deploy Wheel Mode?(1)

7. If the system is in Orbit Night Mode, what is the minimum number of transitions necessary to enter Reorient Mode?(2)

8. If the system is in Acquire Mode, and has been in this mode *less* than the Acquire Mode Delay time, is it possible to transition to Spinup Mode? If so, how? (3)

(Going-to specification only)

9. If the system is in Deploy Wheel Mode, to which states can it transition directly?(1)

10. What conditions will trigger a transition from Spinup to Reorient?(2)

11. If the system is in Paddle Deploy Mode, what is the minimum number of transitions necessary to enter Orbit Day Mode?(2)

12. If the system is in Reorient Mode, and the wheel spin rate is greater than the nominal wheel rate, is it possible for the system to transition to Deploy Wheel Mode?(3)

Perspectives Evaluation

1. In general, which perspective did you find easier to read? Did one perspective provide a more intuitive way for you to think about state machine behavior?

2. Do you think making both persectives available is worthwhile, or would you prefer the simplicity of only using a single perspective?

Appendix B: Representations Experiment Specifications

Satellite Control Specification: Textual Representation

Wait Mode

Transition from Wait Mode to Wait Mode if the time that the system has been in the Wait Mode is < Wait Mode Delay.

Transition from Wait Mode to Detumble Mode if the time that the system has been in the Wait Mode is ≥ the Wait Mode Delay.

Detumble Mode

Transition from Detumble Mode to Detumble Mode if either

1. the time that the system has been in Detumble Mode is < Detumble Mode Delay, OR

2. time that the system has been in Detumble Mode is ≥ Detumble Mode Delay and the XZ Momentum Error is > XZ Momentum Error Threshold.

Transition from Detumble Mode to Spinup Mode if the time that the system has been in the Detumble Mode is ≥ Detumble Mode Delay, and the XZ Momentum Error is ≤ XZ Momentum Error Threshold.

Spinup Mode

Transition from Spinup Mode to Orbit Night Mode if the Paddles are deployed and the Optical System is tracking.

Transition from Spinup Mode to Spinup Mode if either

1. the paddles are not deployed or the Optical System is not tracking, AND the time that the system has been in the Spinup Mode is < Spinup Mode Delay, OR

2. the paddles are not deployed or the Optical System is not tracking, AND the time that the system has been in the Spinup Mode is ≥ Spinup Mode Delay, AND the XZ Momentum Error ≤ XZ Momentum Error Threshold, AND the Momentum Error > Spinup Momentum Error or the system is in Eclipse.

Transition from Spinup Mode to Detumble Mode if the paddles are not deployed or the Optical Sytem is not tracking , AND the time that the system has been in Spinup Mode ≥ Spinup Mode Delay, AND XZ Momentum Error > XZ Momentum Error Threshold.

Transition from Spinup Mode to Reorient Mode if the paddles are not deployed or the Optical System is not tracking, AND the time that the system has been in Spinup Mode ≥ Spinup Mode Delay, AND XZ Momentum Error ≤ XZ Momentum Error Threshold, AND the momentum error ≤ Spinup Momentum Error, AND the system is not in Eclipse.

Reorient Mode

Transition from Reorient Mode to Spinup Mode if either

1. the system is in Eclipse, OR

2. the system is not in Eclipse, AND the time that the system has been in Reorient Mode is ≥ Reorient Mode Delay, AND the momentum error > Spinup Momentum Error.

Transition from Reorient Mode to Reorient Mode if either

1. the system is not in Eclipse, AND the time that the system has been in Reorient Mode is < Reorient Mode Delay

2. the system is not in Eclipse, AND the time that the system has been in Reorient Mode is ≥ Reorient Mode Delay, AND momentum error ≤ Spinup Momentum Error, AND Sine Sun Elevation > Coarse Sun Elevation Error.

Transition from Reorient Mode to Deploy Wheel Mode if the system is not in Eclipse, AND the time that the sytem has been in Reorient Mode is ≥ Reorient Mode Delay, AND the momentum error ≤ Spinup Momentum Error, AND Sine Sun Elevation ≤ Coarse Sun Elevation Error, AND the wheel spin rate < Nominal Wheel Rate.

Transition from Reorient Mode to Acquire Mode if the system is not in Eclipse, AND the time that the system has been in Reorient Mode is ≥ Reorient Mode Delay, AND the momentum error ≤ Spinup Momentum Error, AND Sine Sun Elevation ≤ Coarse Sun Elevation Error, AND the wheel spin rate ≥ Nominal Wheel Rate.

Deploy Wheel Mode

Transition from Deploy Wheel Mode to Deploy Wheel Mode if the time that the system has been in Deploy Wheel Mode is < Deploy Wheel Mode Delay.

Transition from Deploy Wheel Mode to Acquire Mode if the time that the system has been in Deploy Wheel Mode is ≥ Deploy Wheel Mode Delay.

Acquire Mode

Transition from Acquire Mode to Spinup Mode if the system is in Eclipse.

Transition from Acquire Mode to Acquire Mode if either

1. the system is not in Eclipse, AND the time that the system has been in Acquire Mode ≥ Acquire Mode Delay, AND Sine Sun Elevation > Coarse Sine Sun Elevation Error.

2. the system is not in Eclipse, AND the time that the system has been in Acquire Mode ≥ Acquire Mode Delay, AND Sine Sun Elevation ≤ Coarse Sin Sun Elevation Error, AND Sine Sun Azimuth > Coarse Sun Azimuth Error.

Transition from Acquire Mode to Reorient Mode if the system is not in Eclipse, AND the time that the system has been in Acquire Mode ≥ Acquire Mode Delay, AND Sine Sun Elevation > Coarse Sin Sun Elevation Error.

Transition from Acquire Mode to Paddle Deploy Mode if the system is not in Eclipse, AND the time that the system has been in Acquire Mode ≥ Acquire Mode Delay, AND Sine Sun Elevation ≤ Coarse Sin Sun Elevation Error, AND Sine Sun Azimuth ≤ Coarse Sun Azimuth Error, AND the paddles are not deployed.

Transition from Acquire Mode to Orbit Day Mode if the system is in Eclipse, AND the time that the system has been in Acquire Mode ≥ Acquire Mode Delay, AND Sine Sun Elevation ≤ Coarse Sin Sun Elevation Error, AND Sine Sun Azimuth ≤ Coarse Sun Azimuth Error, AND the paddles are deployed.

Paddle Deploy Mode

Transition from Paddle Deploy Mode to Acquire Mode if either

1. the paddles are deployed, OR
2. the paddles are not deployed, AND the time that the system has been in Paddle Deploy Mode ≥ Paddle Deploy Mode Delay

Transition from Paddle Deploy Mode to Paddle Deploy Mode if the paddles are not deployed, AND the time that the system has been in Paddle Deploy Mode < Paddle Deploy Mode Delay.

Orbit Day Mode

Transition from Orbit Day Mode to Spinup Mode if the system is in Eclipse, AND the Optical System is not tracking.

Transition from Orbit Day Mode to Orbit Day Mode if either

1. the system is not in Eclipse or the Optical System is tracking, AND the time that the system has been in Orbit Delay Mode < Orbit Day Mode Delay, OR

2. the system is not in Eclipse or the Optical System is tracking, AND the time that the system has been in Orbit Delay Mode ≥ Orbit Day Mode Delay, AND the Sine Sun Elevation ≤ Coarse Sine Sun Elevation Error, AND the Sine Sun Azimuth ≤ the Coarse Sine Sun Azimuth Error, AND the Optical System is not Tracking.

Transition from Orbit Day Mode to Reorient Mode if the system is not in Eclipse or the Optical System is Tracking, AND the time that the system has been in Orbit Day Mode is ≥ Orbit Day Mode Delay, AND Sine Sun Elevation > Coarse Sine Sun Elevation Error.

Transition from Orbit Day Mode to Acquire Mode if the system is not in Eclipse or the Optical System is tracking, AND the time that the system has been in Orbit Day Mode is ≥ Orbit Day Mode Delay, AND Sine Sun Elevation ≤ Coarse Sine Sun Elevation Error, AND Sine Sun Azimuth > Coarse Sine Sun Azimuth Error.

Transition from Orbit Day Mode to Orbit Night Mode if the system is either not in Eclipse or the Optical System is tracking, AND the time that the system has been in Orbit Day Mode is ≥ Orbit Day Mode Delay, AND Sine Sun Elevation ≤ Coarse Sine Sun Elevation Error, AND Sine Sun Azimuth ≤ Coarse Sine Sun Azimuth Error, AND the Optical System is tracking.

Orbit Night Mode

Transition from Orbit Night Mode to Orbit Day Mode if the system is not in Eclipse, AND either

1. Sine Sun Azimuth > Fine Azimuth Error, OR
2. Sine Sun Elevation > Fine Elevation Error, OR
3. the Optical System is not tracking.

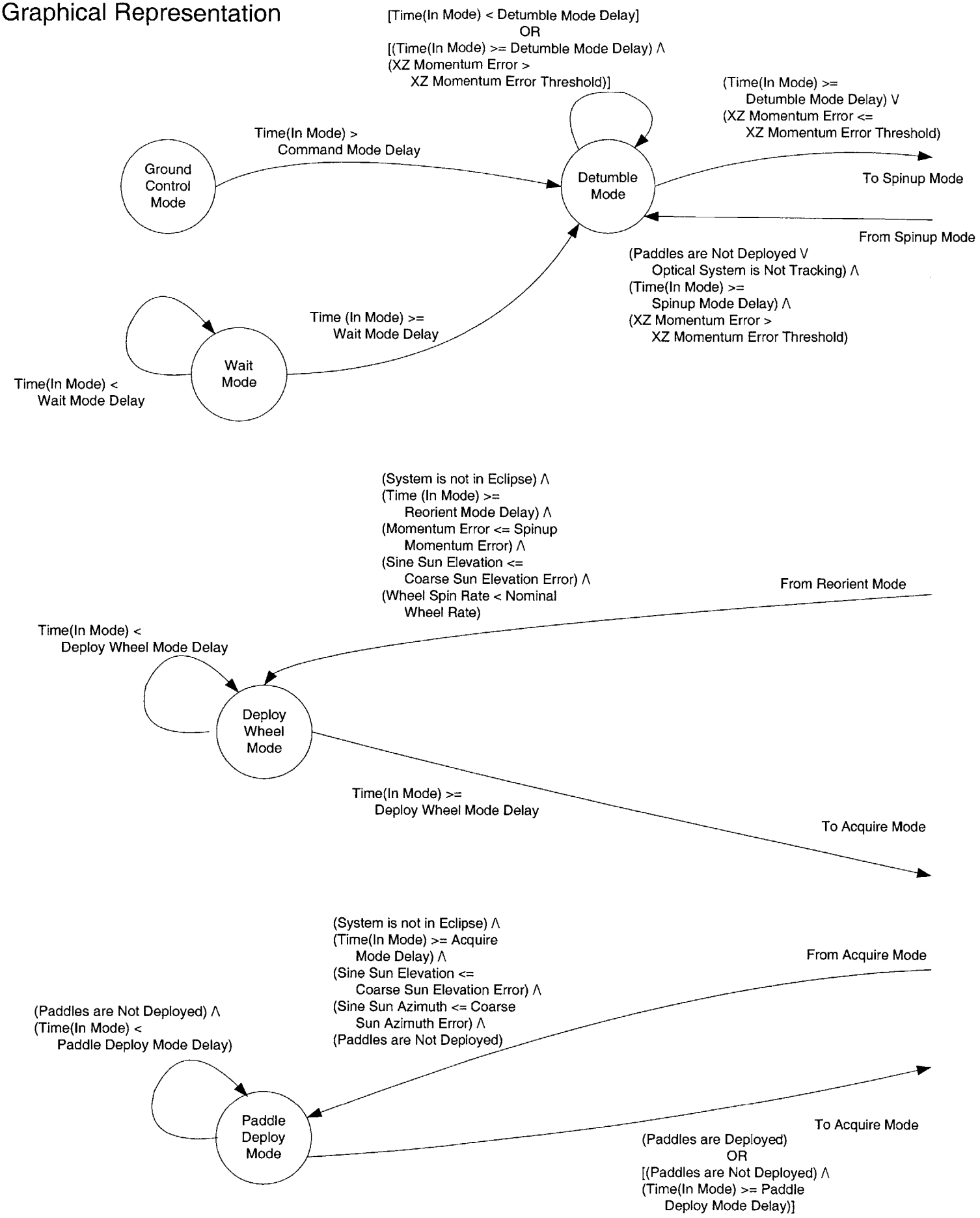
Transition from Orbit Night Mode to Spinup Mode if the system is in Eclipse, AND Sine Sun Azimuth ≤ Fine Azimuth Error, AND Sine Sun Elevation ≤ Fine Elevation Error, AND the Optical System is tracking, AND ω > Maximum Tumble Rate.

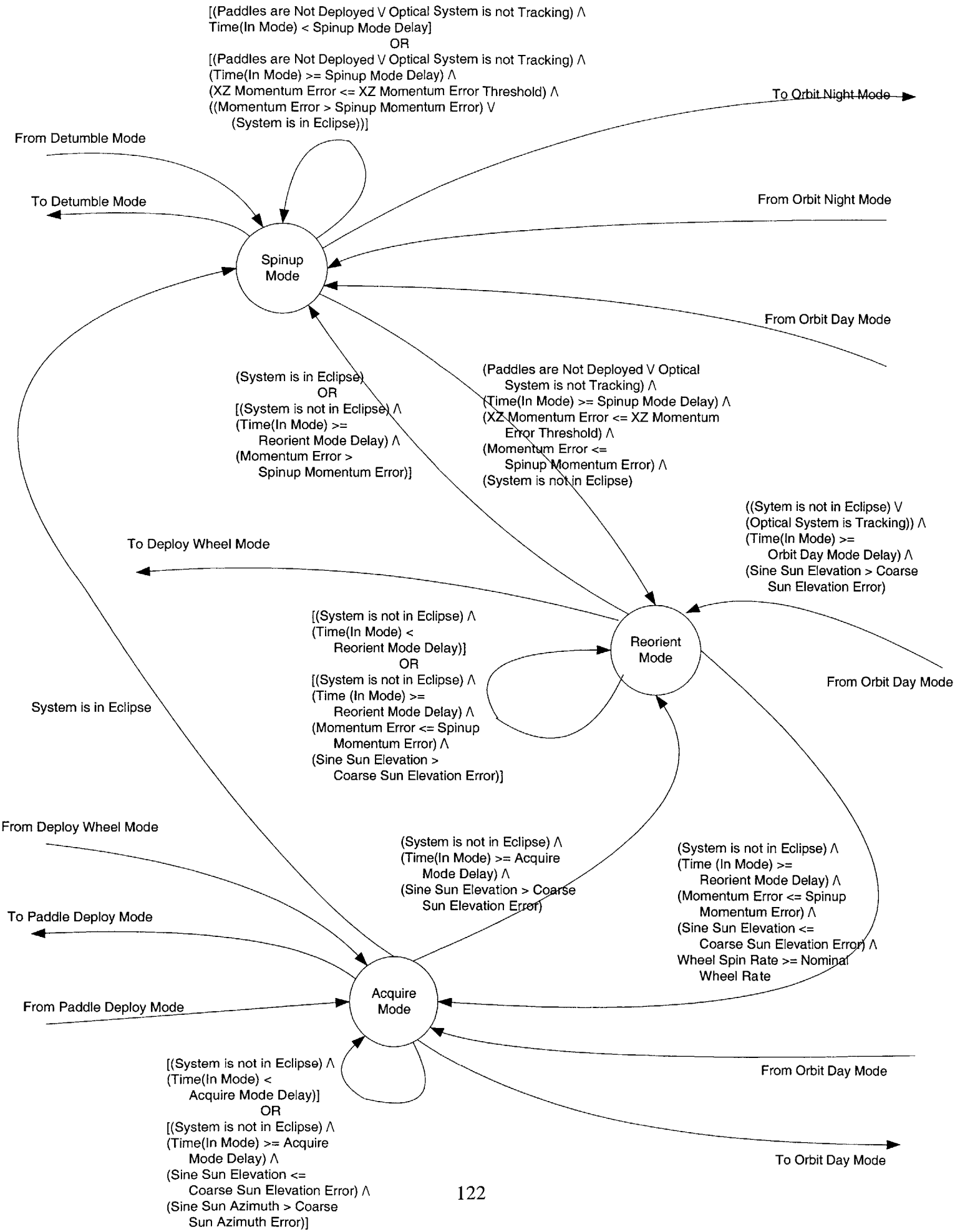
Transition from Orbit Night Mode to Orbit Night Mode if the system is in Eclipse, AND Sine Sun Azimuth ≤ Fine Azimuth Error, AND Sine Sun Elevation ≤ Fine Elevation Error, AND the Optical System is tracking, AND $\omega \leq$ Maximum Tumble Rate.

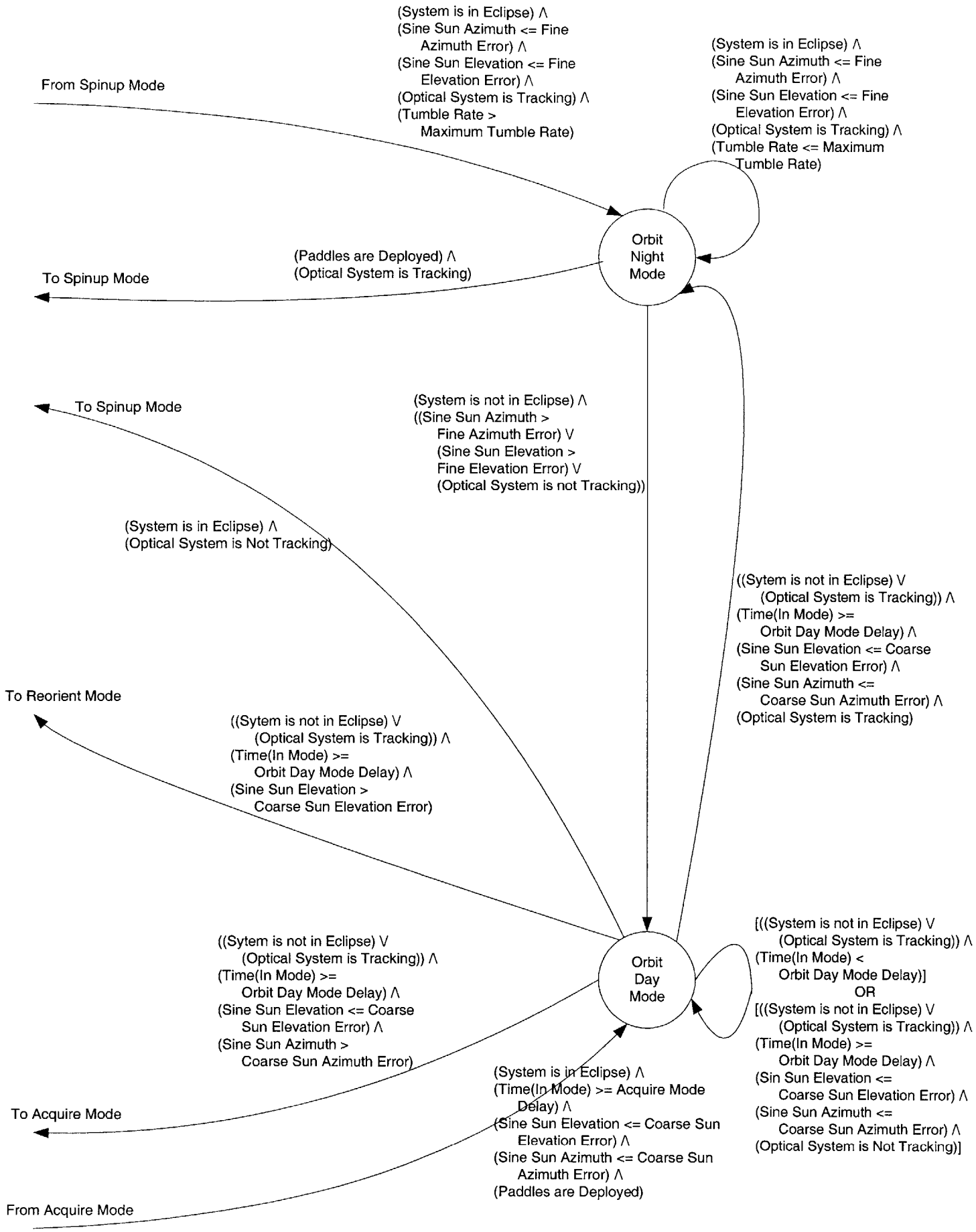
Ground Control Mode

Transition from Ground Control Mode to Detumble Mode if the time that the system has been in Ground Control Mode is > Command Mode Delay.

Satellite Control Specification: Graphical Representation




[(Paddles are Not Deployed V Optical System is not Tracking) Λ
Time(In Mode) < Spinup Mode Delay]
OR
[(Paddles are Not Deployed V Optical System is not Tracking) Λ
(Time(In Mode) >= Spinup Mode Delay) Λ
(XZ Momentum Error <= XZ Momentum Error Threshold) Λ
((Momentum Error > Spinup Momentum Error) V
(System is in Eclipse))]
To Orbit Night Mode
From Detumble Mode
To Detumble Mode
From Orbit Night Mode
Spinup Mode
From Orbit Day Mode
(System is in Eclipse)
OR
[(System is not in Eclipse) Λ
(Time(In Mode) >=
Reorient Mode Delay) Λ
(Momentum Error >
Spinup Momentum Error)]
(Paddles are Not Deployed V Optical
System is not Tracking) Λ
(Time(In Mode) >= Spinup Mode Delay) Λ
(XZ Momentum Error <= XZ Momentum
Error Threshold) Λ
(Momentum Error <=
Spinup Momentum Error) Λ
(System is not in Eclipse)
((Sytem is not in Eclipse) V
(Optical System is Tracking)) Λ
(Time(In Mode) >=
Orbit Day Mode Delay) Λ
(Sine Sun Elevation > Coarse
Sun Elevation Error)
To Deploy Wheel Mode
[(System is not in Eclipse) Λ
(Time(In Mode) <
Reorient Mode Delay)]
OR
[(System is not in Eclipse) Λ
(Time (In Mode) >=
Reorient Mode Delay) Λ
(Momentum Error <= Spinup
Momentum Error) Λ
(Sine Sun Elevation >
Coarse Sun Elevation Error)]
Reorient Mode
From Orbit Day Mode
System is in Eclipse
From Deploy Wheel Mode
(System is not in Eclipse) Λ
(Time(In Mode) >= Acquire
Mode Delay) Λ
(Sine Sun Elevation > Coarse
Sun Elevation Error)
(System is not in Eclipse) Λ
(Time (In Mode) >=
Reorient Mode Delay) Λ
(Momentum Error <= Spinup
Momentum Error) Λ
(Sine Sun Elevation <=
Coarse Sun Elevation Error) Λ
Wheel Spin Rate >= Nominal
Wheel Rate
To Paddle Deploy Mode
Acquire Mode
From Paddle Deploy Mode
[(System is not in Eclipse) Λ
(Time(In Mode) <
Acquire Mode Delay)]
OR
[(System is not in Eclipse) Λ
(Time(In Mode) >= Acquire
Mode Delay) Λ
(Sine Sun Elevation <=
Coarse Sun Elevation Error) Λ
(Sine Sun Azimuth > Coarse
Sun Azimuth Error)]
From Orbit Day Mode
To Orbit Day Mode


From Spinup Mode
(System is in Eclipse) Λ
(Sine Sun Azimuth <= Fine
Azimuth Error) Λ
(Sine Sun Elevation <= Fine
Elevation Error) Λ
(Optical System is Tracking) Λ
(Tumble Rate >
Maximum Tumble Rate)
(System is in Eclipse) Λ
(Sine Sun Azimuth <= Fine
Azimuth Error) Λ
(Sine Sun Elevation <= Fine
Elevation Error) Λ
(Optical System is Tracking) Λ
(Tumble Rate <= Maximum
Tumble Rate)
Orbit
Night
Mode
(Paddles are Deployed) Λ
(Optical System is Tracking)
To Spinup Mode
To Spinup Mode
(System is not in Eclipse) Λ
((Sine Sun Azimuth >
Fine Azimuth Error) V
(Sine Sun Elevation >
Fine Elevation Error) V
(Optical System is not Tracking))
(System is in Eclipse) Λ
(Optical System is Not Tracking)
((Sytem is not in Eclipse) V
(Optical System is Tracking)) Λ
(Time(In Mode) >=
Orbit Day Mode Delay) Λ
(Sine Sun Elevation <= Coarse
Sun Elevation Error) Λ
(Sine Sun Azimuth <=
Coarse Sun Azimuth Error) Λ
(Optical System is Tracking)
To Reorient Mode
((Sytem is not in Eclipse) V
(Optical System is Tracking)) Λ
(Time(In Mode) >=
Orbit Day Mode Delay) Λ
(Sine Sun Elevation >
Coarse Sun Elevation Error)
((Sytem is not in Eclipse) V
(Optical System is Tracking)) Λ
(Time(In Mode) >=
Orbit Day Mode Delay) Λ
(Sine Sun Elevation <= Coarse
Sun Elevation Error) Λ
(Sine Sun Azimuth >
Coarse Sun Azimuth Error)
Orbit
Day
Mode
[((System is not in Eclipse) V
(Optical System is Tracking)) Λ
(Time(In Mode) <
Orbit Day Mode Delay)]
OR
[((System is not in Eclipse) V
(Optical System is Tracking)) Λ
(Time(In Mode) >=
Orbit Day Mode Delay) Λ
(Sin Sun Elevation <=
Coarse Sun Elevation Error) Λ
(Sine Sun Azimuth <=
Coarse Sun Azimuth Error) Λ
(Optical System is Not Tracking)]
To Acquire Mode
(System is in Eclipse) Λ
(Time(In Mode) >= Acquire Mode
Delay) Λ
(Sine Sun Elevation <= Coarse Sun
Elevation Error) Λ
(Sine Sun Azimuth <= Coarse Sun
Azimuth Error) Λ
(Paddles are Deployed)
From Acquire Mode

Satellite Control Specification
Tabular Representation

Current Mode	Wait	
Paddles		
Time In Mode	< Wait Mode Delay	>= Wait Mode Delay
XZ Momentum Error		
Optical System		
Momentum Error		
In Eclipse		
Sine Sun Elevation		
Wheel Spin Rate		
Sine Sun Azimuth		
Tumble Rate, Omega		
New Mode	**Wait**	**Detumble**

Satellite Control Specification
Tabular Representation

Current Mode	Detumble		
Paddles			
Time In Mode	< Detumble Mode Delay	>= Detumble Mode Delay	>= Detumble Mode Delay
XZ Momentum Error		> XZ Momentum Error Threshold	<= XZ Momentum Error Threshold
Optical System			
Momentum Error			
In Eclipse			
Sine Sun Elevation			
Wheel Spin Rate			
Sine Sun Azimuth			
Tumble Rate, Omega			
New Mode	**Detumble**	**Detumble**	**Spinup**

Satellite Control Specification
Tabular Representation

Current Mode	Spinup						
Paddles	Deployed	Not Deployed		Not Deployed		Not Deployed	
Time In Mode		< Spinup Mode Delay	<Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay
XZ Momentum Error				<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold
Optical System	Tracking		Not Tracking		Not Tracking		Not Tracking
Momentum Error				> Spinup Momentum Error	> Spinup Momentum Error		
In Eclipse						TRUE	TRUE
Sine Sun Elevation							
Wheel Spin Rate							
Sine Sun Azimuth							
Tumble Rate, Omega							
New Mode	**Orbit Night**	**Spinup**	**Spinup**	**Spinup**	**Spinup**	**Spinup**	**Spinup**

Satellite Control Specification
Tabular Representation

Current Mode	Spinup (cont.)			
Paddles	Not Deployed		Not Deployed	
Time In Mode	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay
XZ Momentum Error	> XZ Momentum Error Threshold	> XZ Momentum Error Threshold	<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold
Optical System		Not Tracking		Not Tracking
Momentum Error			<= Spinup Momentum Error	<= Spinup Momentum Error
In Eclipse			FALSE	FALSE
Sine Sun Elevation				
Wheel Spin Rate				
Sine Sun Azimuth				
Tumble Rate, Omega				
New Mode	**Detumble**	**Detumble**	**Reorient**	**Reorient**

Satellite Control Specification
Tabular Representation

Current Mode	Reorient					
Paddles						
Time In Mode		>= Reorient Mode Delay	< Reorient Mode Delay	>= Reorient Mode Delay	>= Reorient Mode Delay	>= Reorient Mode Delay
XZ Momentum Error						
Optical System						
Momentum Error		> Spinup Momentum Error		<= Spinup Momentum Error	<= Spinup Momentum Error	<= Spinup Momentum Error
In Eclipse	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
Sine Sun Elevation				> Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error
Wheel Spin Rate					< Nominal Wheel Rate	>= Nominal Wheel Rate
Sine Sun Azimuth						
Tumble Rate, Omega						
New Mode	**Spinup**	**Spinup**	**Reorient**	**Reorient**	**Deploy Wheel**	**Acquire**

Satellite Control Specification
Tabular Representation

Current Mode	Deploy Wheel	
Paddles		
Time In Mode	< Deploy Wheel Mode Delay	>= Deploy Wheel Mode Delay
XZ Momentum Error		
Optical System		
Momentum Error		
In Eclipse		
Sine Sun Elevation		
Wheel Spin Rate		
Sine Sun Azimuth		
Tumble Rate, Omega Omega		
New Mode	**Deploy Wheel**	**Acquire**

Satellite Control Specification
Tabular Representation

Current Mode	Acquire					
Paddles					Not Deployed	Deployed
Time In Mode		< Acquire Mode Delay	>= Acquire Mode Delay	>= Acquire Mode Delay	>= Acquire Mode Delay	>= Acquire Mode Delay
XZ Momentum Error						
Optical System						
Momentum Error						
In Eclipse	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE
Sine Sun Elevation			<= Coarse Sun Elevation Error	> Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error
Wheel Spin Rate						
Sine Sun Azimuth			> Coarse Sun Azimuth Error		<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error
Tumble Rate, Omega Omega						
New Mode	**Spinup**	**Acquire**	**Acquire**	**Reorient**	**Paddle Deploy**	**Orbit Day**

Satellite Control Specification
Tabular Representation

Current Mode	Paddle Deploy		
Paddles	Deployed	Not Deployed	Not Deployed
Time In Mode		>= Paddle Deploy Mode Delay	< Paddle Deploy Mode Delay
XZ Momentum Error			
Optical System			
Momentum Error			
In Eclipse			
Sine Sun Elevation			
Wheel Spin Rate			
Sine Sun Azimuth			
Tumble Rate, Omega Omega			
New Mode	**Acquire**	**Acquire**	**Paddle Deploy**

Satellite Control Specification
Tabular Representation

Current Mode	Orbit Day						
Paddles							
Time In Mode		< Orbit Day Mode Delay	< Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay
XZ Momentum Error							
Optical System	Not Tracking		Tracking	Not Tracking	Tracking		Tracking
Momentum Error							
In Eclipse	TRUE	FALSE		FALSE		FALSE	
Sine Sun Elevation				<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	> Coarse Sun Elevation Error	> Coarse Sun Elevation Error
Wheel Spin Rate							
Sine Sun Azimuth				<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error		
Tumble Rate, Omega							
New Mode	**Spinup**	**Orbit Day**	**Orbit Day**	**Orbit Day**	**Orbit Day**	**Reorient**	**Reorient**

Satellite Control Specification
Tabular Representation

Current Mode	Orbit Day (cont.)			
Paddles				
Time In Mode	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay
XZ Momentum Error				
Optical System		Tracking	Tracking	Tracking
Momentum Error				
In Eclipse	FALSE		FALSE	
Sine Sun Elevation	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error
Wheel Spin Rate				
Sine Sun Azimuth	> Coarse Sun Azimuth Error	> Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error
Tumble Rate, Omega				
New Mode	**Acquire**	**Acquire**	**Orbit Night**	**Orbit Night**

Satellite Control Specification
Tabular Representation

Current Mode	Orbit Night				
Paddles					
Time In Mode					
XZ Momentum Error					
Optical System			Not Tracking	Tracking	Tracking
Momentum Error					
In Eclipse	FALSE	FALSE	FALSE	TRUE	TRUE
Sine Sun Elevation		> Fine Elevation Error		<= Fine Elevation Error	<= Fine Elevation Error
Wheel Spin Rate					
Sine Sun Azimuth	> Fine Azimuth Error			<= Fine Azimuth Error	<= Fine Azimuth Error
Tumble Rate, Omega				> Maximum Tumble Rate	<= Maximum Tumble Rate
New Mode	**Orbit Day**	**Orbit Day**	**Orbit Day**	**Spinup**	**Orbit Night**

Satellite Control Specification
Tabular Representation

Current Mode	Ground Control
Paddles	
Time In Mode	> Command Mode Delay
XZ Momentum Error	
Optical System	
Momentum Error	
In Eclipse	
Sine Sun Elevation	
Wheel Spin Rate	
Sine Sun Azimuth	
Tumble Rate, Omega	
New Mode	**Detumble**

Appendix C: Conditions Experiment Specifications

Climb FMS Speed Mode
Tabular Specification

= Default **IF**

PREV(Flight Phase) = Done	F	*	*	*	*	*	*	*	*
Flight Phase = Done	T	*	*	*	*	*	*	*	*
PREV(Flight Phase) = Takeoff	*	T	*	*	*	*	*	*	*
Flight Phase = Descent	*	T	*	T	*	*	*	*	*
PREV(Flight Phase) = Climb	*	*	T	T	*	*	*	*	*
Flight Phase = Cruise	*	*	T	*	*	*	*	*	*
Climb FMS Speed Mode = Max Climb	*	*	*	*	T	T	T	*	*
FCC Engaged Mode = Altitude Hold Speed	*	*	*	*	T	*	*	*	*
FCC Engaged Mode = Altitude Hold Idle Thrust	*	*	*	*	*	T	*	*	*
FCC Engaged Mode = Altitude Hold Maximum Thrust	*	*	*	*	*	*	T	*	*
PREV(Engine Out) = Not Engaged	*	*	*	*	*	*	*	T	*
Engine Out = Engaged	*	*	*	*	*	*	*	T	*
FMS Mode = Lateral Only	*	*	*	*	*	*	*	*	T

= Economy **IF**

Requested FCC Speed Mode = Economy	T	T	T	*	*	*	*	*
Flight Phase = Preflight	T	*	*	T	*	*	*	*
Flight Phase = Takeoff	*	T	*	*	T	*	T	*
Flight Phase = Climb	*	*	T	*	*	T	*	T
Requested FCC Speed Mode = AFS Speed	*	*	*	T	T	T	*	*
Requested Climb Speed Mode	*	*	*	*	*	*	T	T

= Max Climb **IF**

Requested Climb FMS Speed Mode = Max Climb	T

= Edit **IF**

Requested FCC Speed Mode = Edit CAS	T	T	T	*	*	*	*	*	*	*	*	*	
Flight Phase = Preflight	T	*	*	T	*	*	*	*	*	*	*	*	
Flight Phase = Takeoff	*	T	*	*	T	*	*	*	T	*	T	*	
Flight Phase = Climb	*	*	T	*	*	T	*	T	*	T	*	T	
Requested FCC Speed Mode = Edit Mach	*	*	*	T	T	T	*	*	*	*	*	*	
Requested Climb FMS Speed Mode = Edit	*	*	*	*	*	*	T	*	*	*	*	*	
PREV(Flight Phase) = Cruise	*	*	*	*	*	*	*	T	*	*	*	*	
PREV(Climb FMS Speed Mode) = Economy Mode	*	*	*	*	*	*	*	T	T	T	T	T	
PREV(Cruise FMS Speed Mode) = Edit Mode	*	*	*	*	*	*	*	T	*	*	*	*	
PREV(Descent FMS Speed Mode) = Edit Mode	*	*	*	*	*	*	*	*	T	T	T	T	
PREV(Flight Phase) = Descent	*	*	*	*	*	*	*	*	T	T	*	*	
PREV(Flight Phase) = Approach	*	*	*	*	*	*	*	*	*	*	T	T	

Climb FMS Speed Mode
Textual Specification

The Climb FMS Speed Mode shall be the **Default** if any of the following scenarios are true:

1. the Flight Phase transitions to Done

2. the Flight Phase transitions from Takeoff to Descent

3. the Flight Phase transitions from Climb to Cruise

4. the Flight Phase transitions from Climb to Descent

5. the Climb FMS Speed Mode is Max Climb
   AND
   at least one of the following is true:
   a. FCC Engaged Mode is Altitude Hold Speed
   b. FCC Engaged Mode is Altitude Hold Idle Thrust
   c. FCC Engaged Mode is Altitude Hold Maximum Thrust

6. Engine Out transitions from Not Engaged to Engaged

7. FMS Mode is Lateral Only

The Climb FMS Speed Mode shall be **Economy** if any of the following scenarios are true:

1. Economy is requested for the FCC Speed Mode
   AND
   one of the following is true:
   a. Flight Phase is Preflight
   b. Flight Phase is Takeoff
   c. Flight Phase is Climb

2. AFS Speed is requested for the FCC Speed Mode
   AND
   one of the following is true:
   a. Flight Phase is Preflight
   b. Flight Phase is Takeoff
   c. Flight Phase is Climb

3. Economy is requested for the Climb Speed Mode
   AND
   one of the following is true:
   a. Flight Phase is Takeoff
   b. Flight Phase is Climb

The Climb FMS Speed Mode shall be **Max Climb** if any of the following scenarios is true:

1. Max Climb is requested for the Climb FMS Speed Mode

The Climb FMS Speed Mode shall be **Edit** if any of the following scenarios is true:

1. Edit CAS is requested for the FCC Speed Mode
   AND
   one of the following is true:
   a. Flight Phase is Preflight
   b. Flight Phase is Takeoff
   c. Flight Phase is Climb

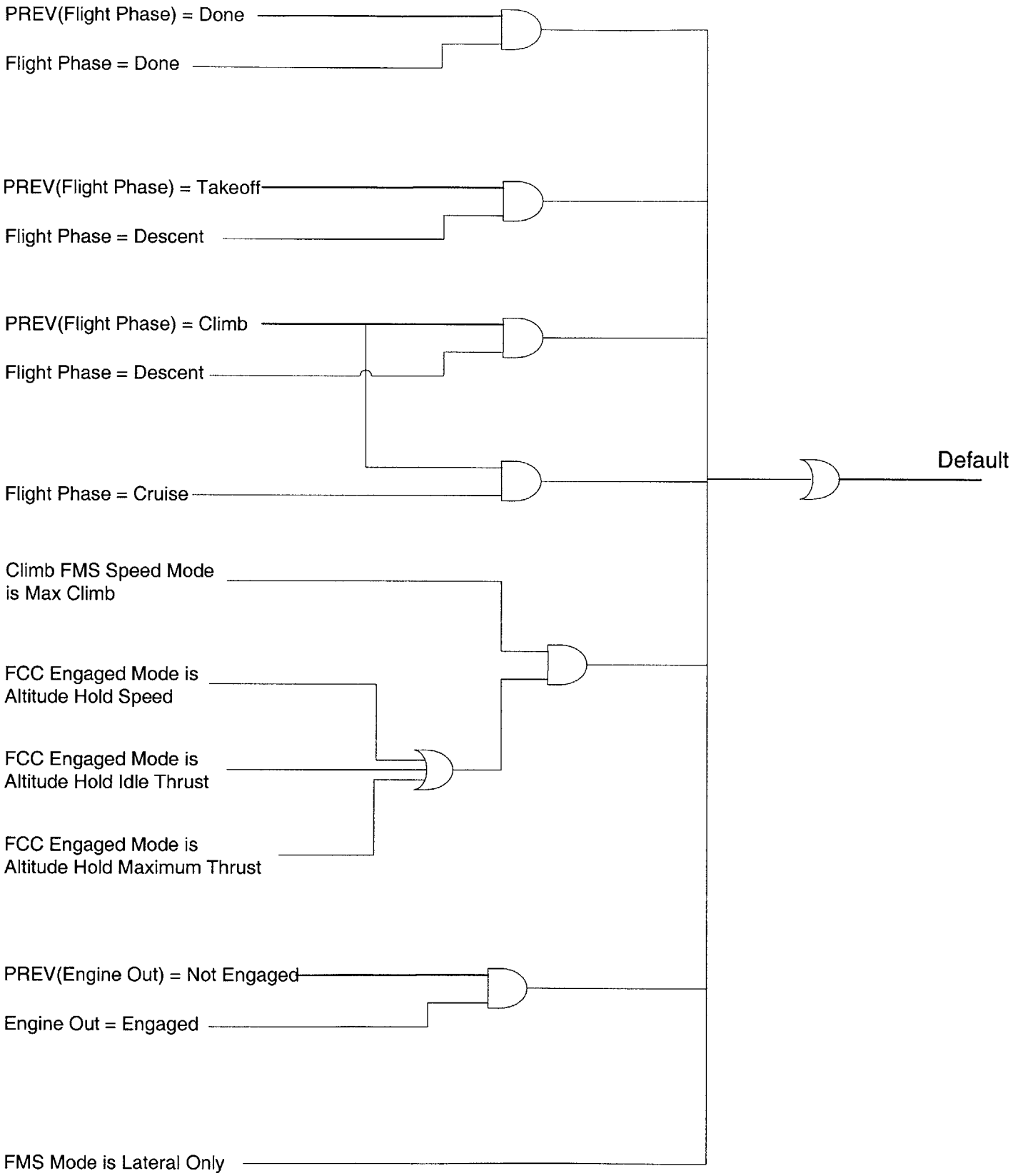
2. Edit Mach is requested for the FCC Speed Mode
   AND
   one of the following is true:
   a. Flight Phase is Preflight
   b. Flight Phase is Takeoff
   c. Flight Phase is Climb

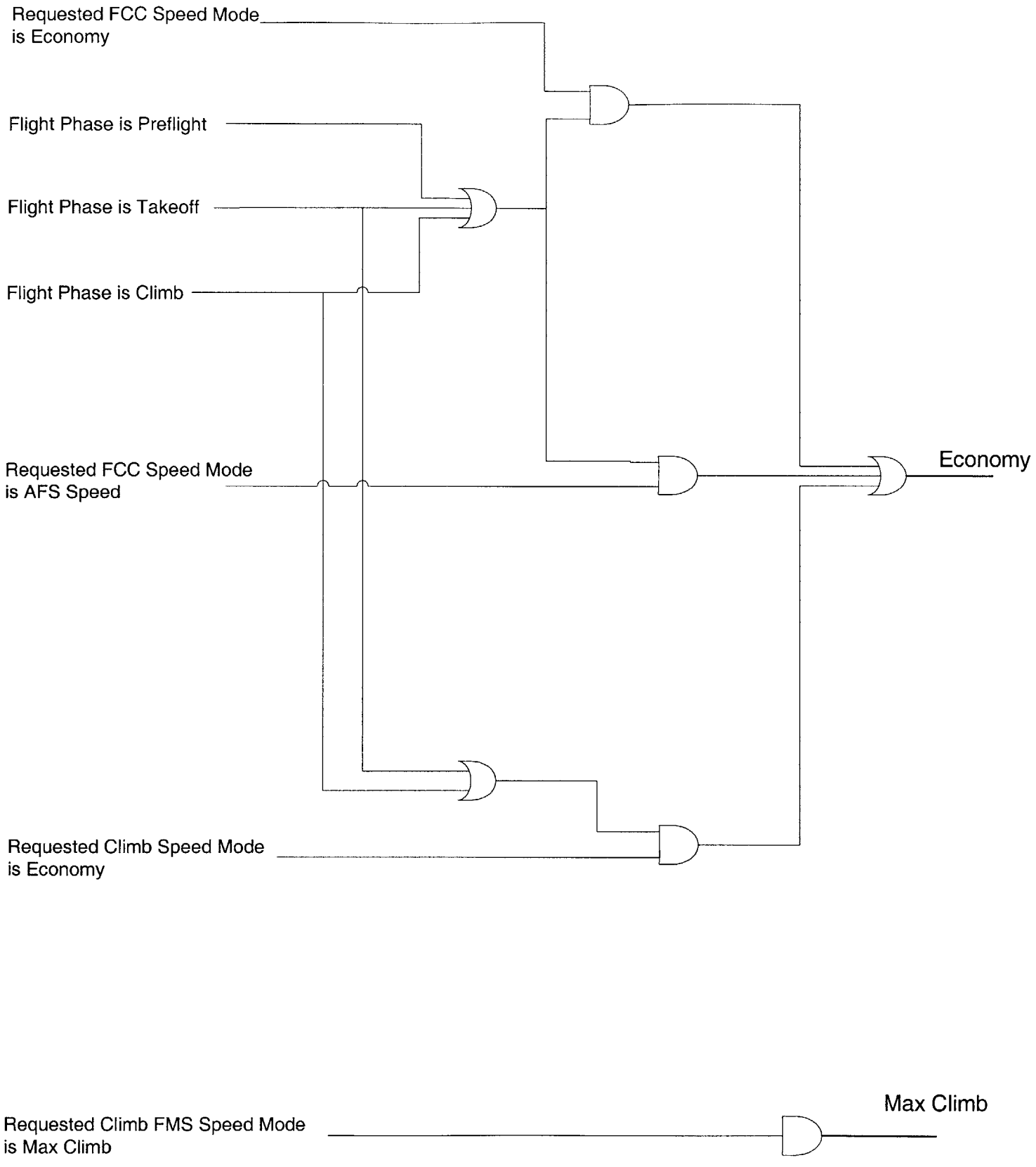
3. Edit is requested for Climb Speed Mode

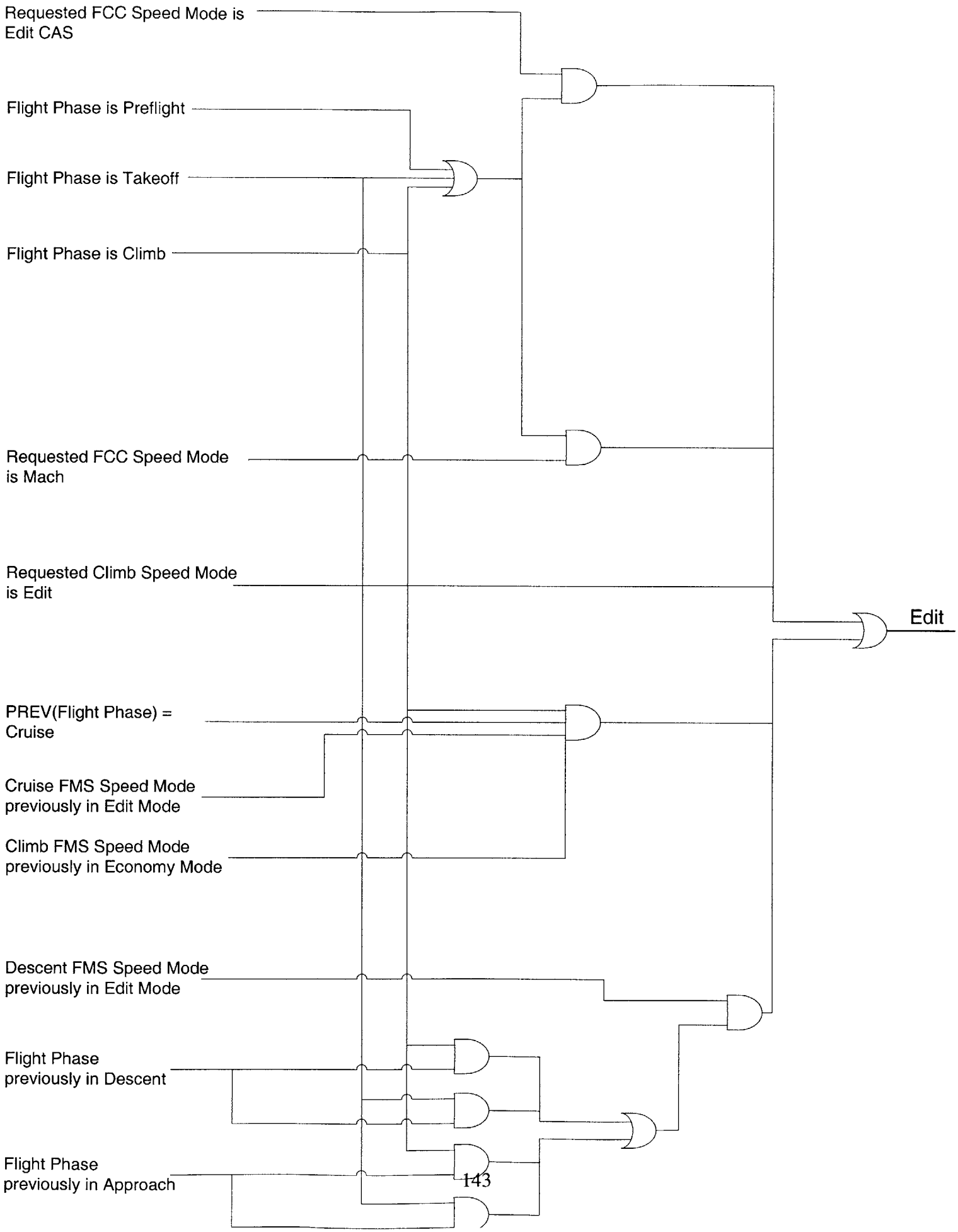
4. Flight Phase transitions from Cruise to Climb
   AND
   Climb FMS Speed Mode previously in Economy Mode
   AND
   Cruise FMS Speed Mode previously in Edit Mode

5. Climb FMS Speed Mode previously in Economy Mode
   AND
   Descent FMS Speed Mode previously in Edit Mode
   AND
   one of the following is true:
   a. Flight Phase transitions from Descent to Takeoff
   b. Flight Phase transitions from Descent to Climb
   c. Flight Phase transitions from Approach to Takeoff
   d. Flight Phase transitions from Approach to Climb

Climb FMS Speed Mode: AND/OR Gate Specification




Requested FCC Speed Mode is Economy
Flight Phase is Preflight
Flight Phase is Takeoff
Flight Phase is Climb
Requested FCC Speed Mode is AFS Speed
Economy
Requested Climb Speed Mode is Economy
Requested Climb FMS Speed Mode is Max Climb
Max Climb


Requested FCC Speed Mode is Edit CAS
Flight Phase is Preflight
Flight Phase is Takeoff
Flight Phase is Climb
Requested FCC Speed Mode is Mach
Requested Climb Speed Mode is Edit
Edit
PREV(Flight Phase) = Cruise
Cruise FMS Speed Mode previously in Edit Mode
Climb FMS Speed Mode previously in Economy Mode
Descent FMS Speed Mode previously in Edit Mode
Flight Phase previously in Descent
Flight Phase previously in Approach

Climb FMS Speed Mode
Propositional Logic Specification

Default =

```
(PREV(Flight Phase ≠ Done) ∧ (Flight Phase = Done)) ∨
(PREV(Flight Phase = Takeoff) ∧ (Flight Phase = Descent)) ∨
(PREV(Flight Phase = Climb) ∧ (Flight Phase = Cruise)) ∨
(PREV(Flight Phase = Climb) ∧ (Flight Phase = Descent)) ∨
((Climb FMS Speed Mode = Max Climb) ∧
 ((FCC Engaged Mode = Altitude Hold Speed) ∨
  (FCC Engaged Mode = Altitude Hold Idle Thrust) ∨
  (FCC Engaged Mode = Altitude Hold Maximum Thrust))) ∨
(PREV(Engine Out = Not Engaged) ∧ (Engine Out = Engaged)) ∨
(FMS Mode = Lateral Only)
```

Economy =

```
((Requested FCC Speed Mode = Economy) ∧
 ((Flight Phase is Preflight) ∨ (Flight Phase is Takeoff) ∨
  (Flight Phase is Climb))) ∨
((Requested FCC Speed Mode = AFS Speed) ∧
 ((Flight Phase is Preflight) ∨
  (Flight Phase is Takeoff) ∨ (Flight Phase is Climb))) ∨
((Requested Climb Speed Mode = Economy) ∧
 ((Flight Phase is Takeoff) ∨ (Flight Phase is Climb)))
```

Max Climb =

```
(Requested Climb FMS Speed Mode = Max Climb)
```

Edit =

```
((Requested FCC Speed Mode = Edit CAS) ∧
 ((Flight Phase is Preflight) ∨ (Flight Phase is Takeoff) ∨
  (Flight Phase is Climb))) ∨
((Requested FCC Speed Mode = Edit Mach) ∧
 ((Flight Phase is Preflight) ∨ (Flight Phase is Takeoff) ∨
  (Flight Phase is Climb))) ∨
(Requested Climb Speed Mode = Edit) ∨
((PREV(Flight Phase = Cruise) ∧ (Flight Phase = Climb)) ∧
 PREV(Climb FMS Speed Mode = Economy Mode) ∧
 PREV(Cruise FMS Speed Mode = Edit Mode)) ∨
(PREV(Climb FMS Speed Mode = Economy Mode) ∧
 PREV(Descent FMS Speed Mode = Edit Mode) ∧
 ((PREV(Flight Phase = Descent) ∧ (Flight Phase = Takeoff)) ∨
  (PREV(Flight Phase = Descent) ∧ (Flight Phase = Climb)) ∨
  (PREV(Flight Phase = Approach) ∧ (Flight Phase = Takeoff)) ∨
  (PREV(Flight Phase = Approach) ∧ (Flight Phase = Climb))))
```

Appendix D: Macros Experiment Specifications

Vertical Guidance Reference Altitude Specification (flat)

State Variable

Vertical Guidance Reference Altitude

= Climb Target Altitude **IF**

Engine Out is engaged	T	T	T	T
Aircraft Attained V3	T	*	T	*
Aircraft is not above 2 Engine Maximum Altitude	T	F	T	F
Engine Out Level Penetration = FALSE	T	*	T	*
Engine Out Level Deceleration = TRUE	*	F	*	F
Flight Phase is Takeoff	T	T	*	*
Flight Phase is Climb	*	*	T	T

= Clearance Altitude **IF**

Flight Phase is Takeoff	T	*
Flight Phase is Climb	*	T
Vertical Guidance Type is Airmass AFS	T	T

= Engine Out Driftdown Deceleration Altitude **IF**

Engine Out is engaged	T	T	T	T	T	T	T	T
Aircraft Altitude > Computed 2 Engine Maximum Altitude	T	T	T	T	T	T	T	T
Aircraft Altitude Rate <= 200	T	*	*	*	T	*	*	*
Flight Phase is Cruise	*	T	*	*	*	T	*	*
Flight Phase is Descent	*	*	T	*	*	*	T	*
Flight Phase is Approach	*	*	*	T	*	*	*	T
Vertical Guidance Type is Profile	T	T	T	T	*	*	*	*
Vertical Guidance Type is Airmass PROF	*	*	*	*	T	T	T	T

= Climb Conflict Altitude **IF**

Engine Out is not engaged	T	T	T	T	T	T	T	T
Flight Phase is Takeoff	*	*	*	*	T	T	T	T
Flight Phase is Climb	T	T	T	T	*	*	*	*
Clearance Altitude < Aircraft Altitude - 250 ft.	T	*	T	*	T	*	T	*
Climb Target Altitude < Aircraft Altitude - 250 ft.	*	T	*	T	*	T	*	*
Vertical Guidance Type is Profile	*	*	T	T	*	*	T	T
Vertical Guidance Type is Airmass PROF	T	T	*	*	T	T	*	*
FCC Autopilot is engaged	T	T	T	T	T	T	T	T

= Actual 2 Engine Maximum Altitude **IF**

Engine Out is engaged	T	T	T
Flight Phase is Cruise	T	*	*
Aircraft is above 2 Engine Maximum Altitude	T	T	T
Aircraft Altitude Rate < 200 ft./sec.	T	T	T
Engine Out Level Deceleration = TRUE	T	T	T
Aircraft initiated maneuver to 2 Engine Maximum Altitude	T	T	T
Flight Phase is Approach	*	T	*
Flight Phase is Descent	*	*	T

= Cruise Conflict Altitude **IF**

Flight Phase is Cruise	T	T	T	T
Clearance Altitude < Aircraft Altitude - 250 ft.	T	T	T	T
Cruise Flight Level > Aircraft Altitude + 250 ft.	T	*	T	*
Vertical Guidance Type is Profile	T	T	*	*
Vertical Guidance Type is Airmass PROF	*	*	T	T
FCC Autopilot is engaged	T	T	T	T
Step Climb is Active	*	T	*	T
Descent/Approach Path Capture/Hold Criteria is not satisfied	T	T	T	T

= Descent Target Altitude **IF**

Flight Phase is Cruise	T	T	T	T
Target Altitude at Active Leg Termination = Descent Target Altitude	T	T	T	T
Clearance Altitude < Aircraft Altitude - 250 ft.	F	*	*	*
Vertical Guidance Type is Airmass AFS	T	T	T	T
FCC Autopilot is engaged	*	F	*	*
Descent/Approach Path Capture/Hold Criteria are satisfied	*	*	T	*
Cruise Flight Level > Aircraft Altitude + 250 ft.	*	*	*	F
Step Climb is Active	*	*	*	F

= Descent Speed Limit Altitude **IF**

Flight Phase is Descent	T	*
Descent Speed Limit Violation is TRUE	T	T
Flight Phase is Approach	*	T

= Descent Conflict Altitude **IF**

Descent Speed Limit Violation is FALSE	T	T	T	T	T	T	T	T
Flight Phase is Descent	T	*	T	*	T	*	T	*
Flight Phase is Approach	*	T	*	T	*	T	*	T
Clearance Altitude < Aircraft Altitude + 250 ft.	T	T	*	*	T	T	*	*
Descent Target Altitude < Aircraft Altitude + 250 ft.	*	*	T	T	*	*	T	T
Vertical Guidance Type is Profile	T	T	T	T	*	*	*	*
Vertical Guidance Type is Airmass AFS	*	*	*	*	T	T	T	T
FCC Autopilot is engaged	T	T	T	T	T	T	T	T
Final Approach Prompt is not selected	T	T	T	T	T	T	T	T

= Altitude Constraint at Destination **IF**

Flight Phase is Descent	T	*	T	*	T	*
Flight Phase is Approach	*	T	*	T	*	T
Descent Speed Limit Violation is FALSE	T	T	T	T	T	T
Profile Descent is TRUE	T	T	T	T	T	T
Final Approach Prompt is not selected	T	T	T	T	T	T
Aircraft is not below path approach level-off	T	T	T	T	T	T
Non-precision VFR Approach Type selected	T	T	T	T	T	T
Clearance Altitude < Aircraft Altitude + 250 ft.	F	F	*	*	*	*
Descent Target Altitude < Aircraft Altitude + 250 ft.	F	F	*	*	*	*
Vertical Guidance Type is Profile	*	*	F	F	*	*
Vertical Guidance Type is Airmass PROF	*	*	F	F	*	*
FCC Autopilot is engaged	*	*	*	*	F	F

= Below Path Approach Level Off Altitude **IF**

Flight Phase is Descent	T	*	T	*	T	*
Flight Phase is Approach	*	T	*	T	*	T
Descent Speed Limit Violation is FALSE	T	T	T	T	T	T
Profile Descent is TRUE	T	T	T	T	T	T
Final Approach Prompt is selected	T	T	T	T	T	T
Aircraft is below path approach level-off	T	T	T	T	T	T
Non-precision VFR Approach Type selected	T	T	T	T	T	T
Clearance Altitude < Aircraft Altitude + 250 ft.	F	F	*	*	*	*
Descent Target Altitude < Aircraft Altitude + 250 ft.	F	F	*	*	*	*
Vertical Guidance Type is Profile	*	*	F	F	*	*
Vertical Guidance Type is Airmass PROF	*	*	F	F	*	*
FCC Autopilot is engaged	*	*	*	*	F	F

Vertical Guidance Reference Altitude Specification (with Macros)

Macro

Climb Conflict Situation

Climbing	T	T
Clearance Altitude < Aircraft Altitude - 250 ft.	T	*
Climb Target Altitude < Aircraft Altitude - 250 ft.	*	T
PROF Mode engaged	T	T
FCC Autopilot is engaged	T	T

Macro

Descent Conflict Situation

Descending	T	T
Clearance Altitude < Aircraft Altitude + 250 ft.	T	*
Descent Target Altitude < Aircraft Altitude + 250 ft.	*	T
PROF Mode engaged	T	T
FCC Autopilot is engaged	T	T
Final Approach Prompt is not selected	T	T

Macro

Cruise Conflict Situtation

Flight Phase is Cruise	T	T
Clearance Altitude < Aircraft Altitude - 250 ft.	T	T
Cruise Flight Level > Aircraft Altitude + 250 ft.	T	*
PROF Mode engaged	T	T
FCC Autopilot is engaged	T	T
Step Climb is Active	*	T
Descent/Approach Path Capture/Hold Criteria are not satisfied	T	T

Macro

PROF Mode Engaged

Vertical Guidance Type is Profile	T	*
Veritcal Guidance Type is Airmass PROF	*	T

Macro

Climbing

Flight Phase is Takeoff	T	*
Flight Phase is Climb	*	T

Macro

Descending

Flight Phase is Descent	T	*
Flight Phase is Approach	*	T

State Variable

Vertical Guidance Reference Altitude

= Climb Target Altitude **IF**

Engine Out is engaged	T	T
Aircraft Attained V3	T	*
Aircraft is not above 2 Engine Maximum Altitude	T	F
Engine Out Level Penetration = FALSE	T	*
Engine Out Level Deceleration = TRUE	*	F
Climbing	T	T

= Clearance Altitude **IF**

Climbing	T
Vertical Guidance Type is Airmass AFS	T

= Engine Out Driftdown Deceleration Altitude **IF**

Engine Out is engaged	T	T	T
Aircraft Altitude is above Computed 2 Engine Maximum Altitude	T	T	T
Aircraft Altitude Rate <= 200	T	*	*
Flight Phase is Cruise	*	T	*
Descending	*	*	T
PROF Mode Engaged	T	T	T

= Climb Conflict Altitude **IF**

Engine Out is not engaged	T
Climb Conflict Situation	T
Climbing	T

= Actual 2 Engine Maximum Altitude **IF**

Engine Out is engaged	T	T
Flight Phase is Cruise	T	*
Aircraft is above 2 Engine Maximum Altitude	T	T
Aircraft Altitude Rate < 200 ft./sec.	T	T
Engine Out Level Deceleration = TRUE	T	T
Aircraft initiated maneuver to 2 Engine Maximum Altitude	T	T
Descending	*	T

= Cruise Conflict Altitude **IF**

Flight Phase is Cruise	T
Cruise Conflict Situation	T

= Descent Target Altitude **IF**

Flight Phase is Cruise	T
Target Altitude at Active Leg Termination = Descent Target Altitude	T
Cruise Conflict Situation	F
Vertical Guidance Type is Airmass AFS	T

= Descent Speed Limit Altitude **IF**

Descending	T
Descent Speed Limit Violation is TRUE	T

= Descent Conflict Altitude **IF**

Descent Speed Limit Violation is FALSE	T
Descending	T
Descent Conflict Situation	T

= Altitude Constraint at Destination **IF**

Descending	T
Descent Speed Limit Violation is FALSE	T
Profile Descent is TRUE	T
Final Approach Prompt is not selected	T
Aircraft is not below path approach level-off	T
Non-precision VFR Approach Type selected	T
Descent Conflict Situation	F

= Below Path Approach Level Off Altitude **IF**

Descending	T
Descent Speed Limit Violation is FALSE	T
Profile Descent is TRUE	T
Final Approach Prompt is selected	T
Aircraft is below path approach level-off	T
Non-precision VFR Approach Type selected	T
Descent Conflict Situation	F

Controlled Device Status

State	Trigger
On	Device Signal = On
Off	Device Signal = Off
Unknown	Device Signal = Obsolete OR Startup OR Controls Reset OR
Fault-Detected	At least 2 seconds have passed since a (Device Signal = On) message was received

System Status

State	Trigger
Operational	Startup, OR Controls Reset
Internal Fault	Internal Fault Detected
Inhibited	Controls Inhibited

Altitude

State	Trigger
Unknown	Startup OR Controls Reset OR [(**Analog Altimeter** = Unknown) AND (**Digital Altimeter** = Unknown)]
Below Threshold	[(**Analog Altimeter** = Valid) AND (Analog Altimeter Value < 2000)] OR [(**Digital Altimeter** = Valid) AND (Digital Altimeter Value < 2000)]
At or Above Threshold	[(**Analog Altimeter** = Valid) AND (Analog Altimeter Value > 2000) AND (**Digital Altimeter** = Valid) AND (Digital Altimeter Value > 2000)] OR [(**Analog Altimeter** = Valid) AND (Analog Altimeter Value > 2000) AND (**Digital Altimeter** = Valid)] OR [**Analog Altimeter** = Valid) AND (**Digital Altimeter** = Valid) AND (Digital Altimeter Value > 2000)] OR
Cannot be Determined	(**Analog Altimeter** = Invalid) AND (**Digital Altimeter** = Invalid)

Analog Altimeter

State	Trigger
Valid	Analog Status = Valid
Invalid	Analog Status = Invalid
Unknown	(Analog Status = Obsolete) OR Startup OR Controls Reset

Digital Altimeter

State	Trigger
Valid	Digital Status = Valid
Invalid	Digital Status = Invalid
Unknown	(Digital Status = Obsolete) OR Startup OR Controls Reset

Power Command Output

State	Trigger
On	(**Altitude** = Below Threshold) AND (**Controlled Device Status** = Off) AND (**System Status** = Operational) AND (**Altitude** = At-or-above Threshold, in previous cycle)
Off	(**Altitude** = Below Threshold) OR (**Controlled Device Status** = On) OR (**System Status** = Operational) OR (**Altitude** = At-or-above threshold, in previous cycle)

Watchdog Probe Output

State	Trigger
On	[(No more than 200ms have passed since last output was generated) AND (**System Status** = Operational) AND (**Controlled Device Status** = Fault-Detected) AND (At least 2 sec have passed since **Altitude** last entered state Cannot Be Determined)] OR [(No more than 200ms have passed since last output was generated) AND (**System Status** = Inhibited)]

Controlled Device Status

To State	Trigger (Event)	(Condition)	Output Event
On		Device Signal = On	DEVICE ON
Off		Device Signal = Off	DEVICE OFF
Unknown		Device Signal = Obsolete, OR Startup, OR Controls Reset	DEVICE UNKNOWN
Fault-Detected		At least 2 seconds have passed since a (Device Signal = On) message was received	DEVICE FAULT-DETECTED

System Status

State	Trigger (Event)	(Condition)	Output Event
Operational		Startup, OR Controls Reset	SYSTEM OPERATIONAL
Internal Fault		Internal Fault Detected	SYSTEM FAULT
Inhibited		Controls Inhibited	SYSTEM INHIBITED

Altitude

State	Trigger (Event)	(Condition)	Output Event
Unknown		Startup, OR Controls Reset	ALTITUDE UNKNOWN
	ANALOG ALT UNKNOWN	(**Digital Altimeter** = Unknown)	
	DIGITAL ALT UNKNOWN	(**Analog Altimeter** = Unknown)	
Below Threshold	ANALOG ALT VALID	Analog Altimeter Value < 2000	ALTITUDE BELOW
	DIGITAL ALT VALID	Digital Altimeter Value < 2000	
At or Above Threshold	ANALOG ALT VALID	[(Analog Altimeter Value > 2000) AND (**Digital Altimeter** = Valid) AND (Digital Altimeter Value > 2000)] OR [(Analog Altimeter Value > 2000) AND (**Digital Altimeter** = Valid)]	ALTITUDE ABOVE
	ANALOG ALT INVALID, OR ANALOG ALT UNKNOWN	[(**Digital Altimeter** = Valid) AND (Digital Altimeter Value > 2000)]	
	DIGITAL ALT VALID	[(**Analog Altimeter** = Valid) AND (Analog Altimeter Value > 2000) AND (Digital Altimeter Value > 2000)] OR [(**Analog Altimeter** = Valid) AND (Digital Altimeter Value > 2000)]	
	DIGITAL ALT INVALID, OR DIGITAL ALT UNKNOWN	[(**Analog Altimeter** = Valid) AND (Analog Altimeter Value > 2000)]	
Cannot be Determined	ANALOG ALT INVALID	(**Digital Altimeter** = Invalid)	ALTITUDE INVALID
	DIGITAL ALT INVALID	(**Analog Altimeter** = Invalid)	

Analog Altimeter

State	Trigger (Event)	(Condition)	Output Event
Valid		Analog Status = Valid	ANALOG ALT VALID
Invalid		Analog Status = Invalid	ANALOG ALT INVALID
Unknown		Analog Status = Obsolete, OR Startup, OR Controls Reset	ANALOG ALT UNKNOWN

Digital Altimeter

State	Trigger (Event)	(Condition)	Output Event
Valid		Digital Status = Valid	DIGITAL ALT VALID
Invalid		Digital Status = Invalid	DIGITAL ALT INVALID
Unknown		Digital Status = Obsolete, OR Startup, OR Controls Reset	DIGITAL ALT UNKNOWN

Power Command Output

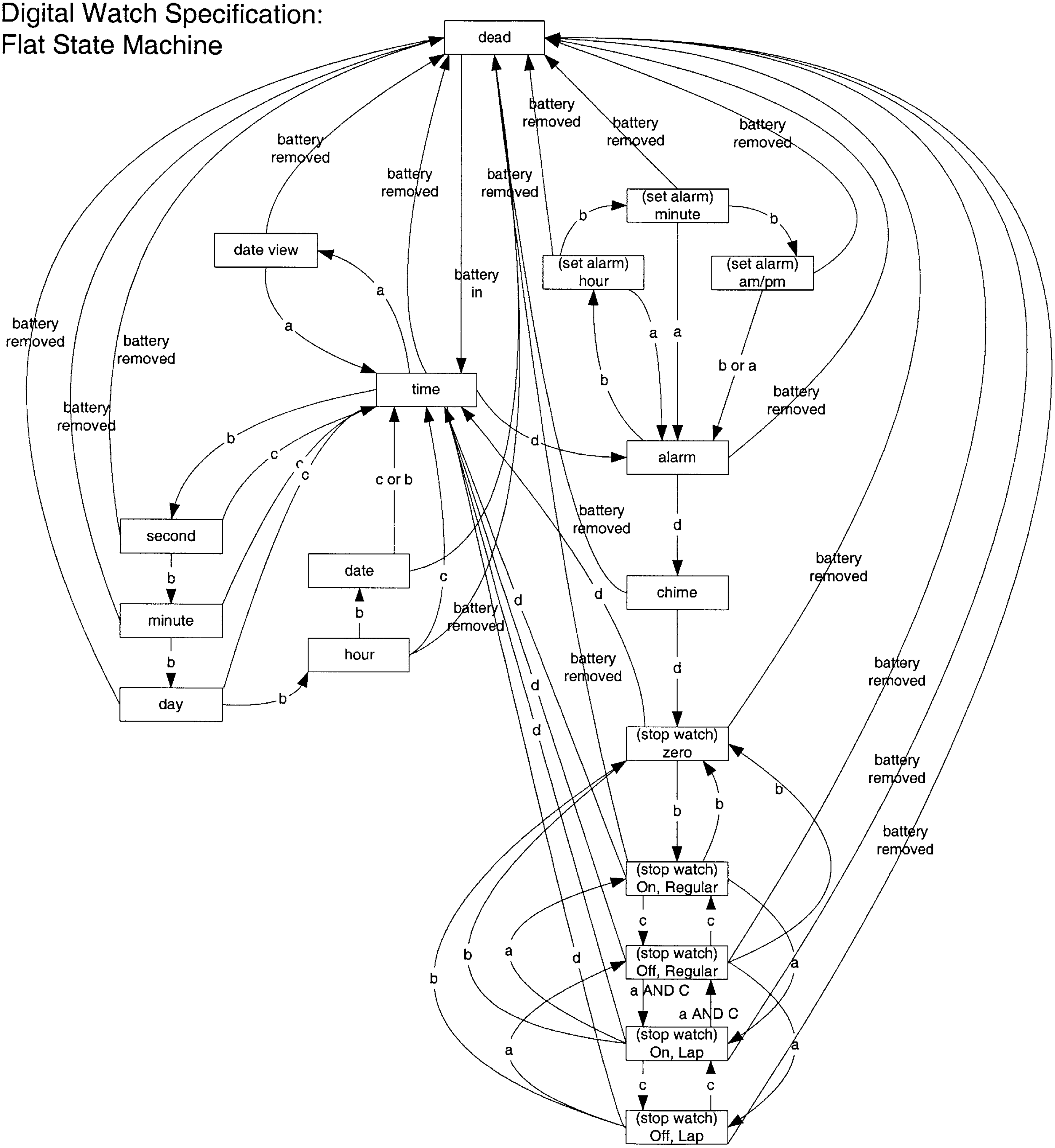
State	Trigger (Event)	(Condition)	Output Event
On	ALTITUDE BELOW	(**Device Status** = Off) AND (**System Status** = Operational) AND (**Altitude** = At-or-above Threshold, in previous cycle)	
Off	ALTITUDE ABOVE, OR ALTITUDE UNKNOWN, OR ALTITUDE INVALID, OR DEVICE ON, OR SYSTEM FAULT, OR SYSTEM INHIBITED		
		(**Altitude** = At-or-above threshold, in previous cycle)	

Watchdog Probe Output

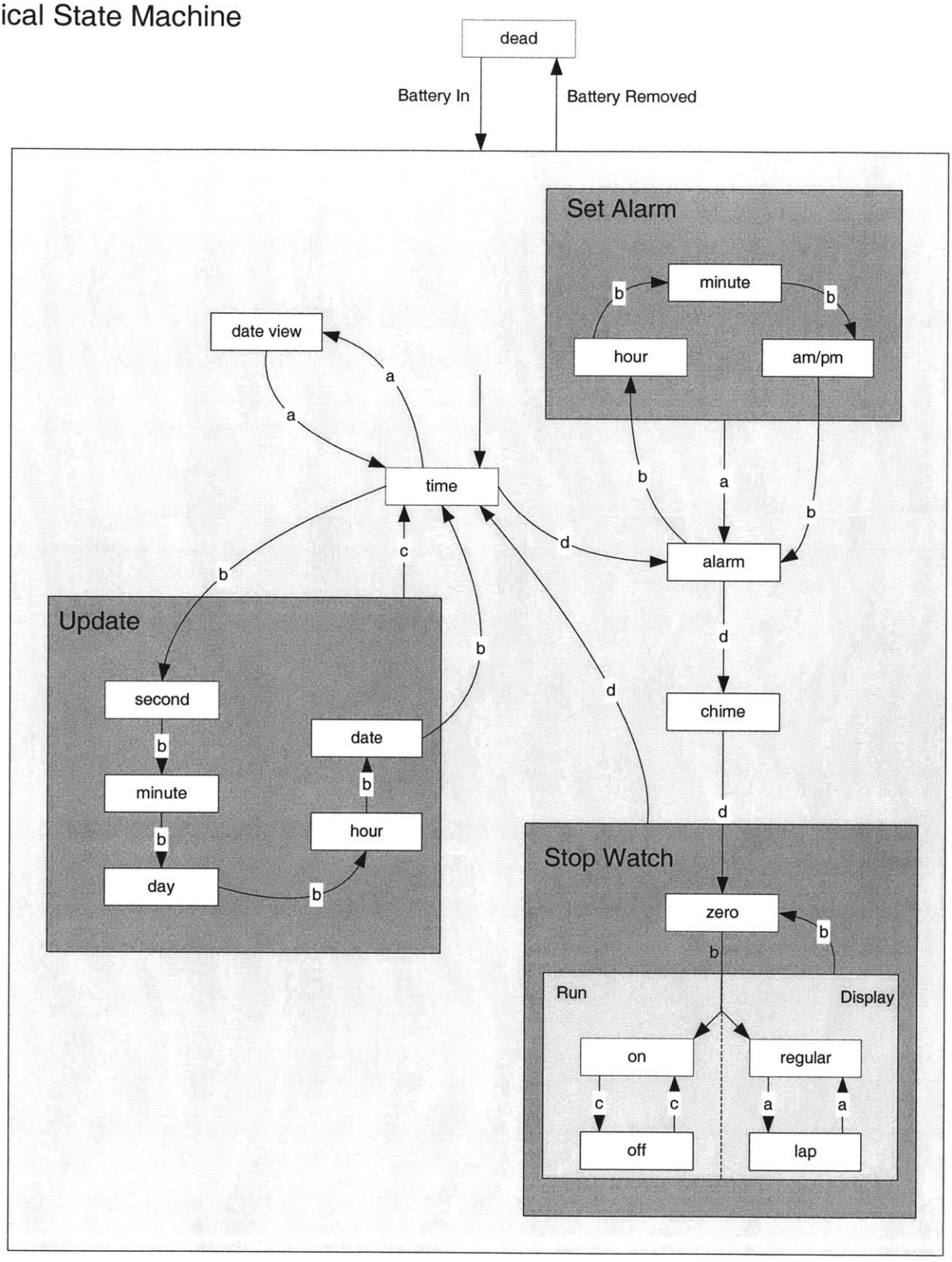
State	Trigger (Event)	(Condition)	Output Event
On		[(No more than 200ms have passed since last output was generated) AND (**System Status** = Operational) AND (**Controlled Device Status** = Fault-Detected) AND (At least 2 sec have passed since **Altitude** last entered state Cannot Be Determined)] OR [(No more than 200ms have passed since last output was generated) AND (**System Status** = Inhibited)]	
	SYSTEM OPERATIONAL	[(No more than 200ms have passed since last output was generated) AND (**Controlled Device Status** = Fault-Detected) AND (At least 2 sec have passed since **Altitude** last entered state Cannot Be Determined)]	
	SYSTEM INHIBITED	No more than 200ms have passed since last output was generated	
	DEVICE FAULT-DETECTED	[(No more than 200ms have passed since last output was generated) AND (**System Status** = Operational) AND (At least 2 sec have passed since **Altitude** last entered state Cannot Be Determined)]	

Appendix F: Hierarchies Experiment Specifications

Digital Watch Specification:
Flat State Machine



Digital Watch Specification:
Hierarchical State Machine



Satellite Control Specification (Coming From Perspective)

Current Mode	Wait
Paddles	
Time In Mode	< Wait Mode Delay
XZ Momentum Error	
Optical System	
Momentum Error	
In Eclipse	
Sine Sun Elevation	
Wheel Spin Rate	
Sine Sun Azimuth	
Tumble Rate, Omega	
New Mode	**Wait**

Satellite Control Specification
(Coming From Perspective)

Current Mode	Wait	Detumble	Detumble	Spinup	Spinup	Ground Control
Paddles				Not Deployed		
Time In Mode	>= Wait Mode Delay	< Detumble Mode Delay	>= Detumble Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay	> Command Mode Delay
XZ Momentum Error			> XZ Momentum Error Threshold	> XZ Momentum Error Threshold	> XZ Momentum Error Threshold	
Optical System					Not Tracking	
Momentum Error						
In Eclipse						
Sine Sun Elevation						
Wheel Spin Rate						
Sine Sun Azimuth						
Tumble Rate, Omega						
New Mode	**Detumble**					

Satellite Control Specification
(Coming From Perspective)

Current Mode	Detumble	Spinup	Spinup	Spinup	Spinup
Paddles		Not Deployed		Not Deployed	
Time In Mode	>= Detumble Mode Delay	< Spinup Mode Delay	<Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay
XZ Momentum Error	<= XZ Momentum Error Threshold			<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold
Optical System			Not Tracking		Not Tracking
Momentum Error				> Spinup Momentum Error	> Spinup Momentum Error
In Eclipse					
Sine Sun Elevation					
Wheel Spin Rate					
Sine Sun Azimuth					
Tumble Rate, Omega					
New Mode	**Spinup**				

Satellite Control Specification
(Coming From Perspective)

Spinup	Spinup	Reorient	Reorient	Acquire	Orbit Day	Orbit Night
Not Deployed						
>= Spinup Mode Delay	>= Spinup Mode Delay		>= Reorient Mode Delay			
<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold					
	Not Tracking				Not Tracking	Tracking
			> Spinup Momentum Error			
TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE
						<= Fine Elevation Error
						<= Fine Azimuth Error
						> Maximum Tumble Rate

Spinup (continued)

Satellite Control Specification
(Coming From Perspective)

Current Mode	Spinup	Spinup	Reorient	Reorient	Acquire	Orbit Day	Orbit Day
Paddles	Not Deployed						
Time In Mode	>= Spinup Mode Delay	>= Spinup Mode Delay	< Reorient Mode Delay	>= Reorient Mode Delay	>= Acquire Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay
XZ Momentum Error	<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold					
Optical System		Not Tracking					Tracking
Momentum Error	<= Spinup Momentum Error	<= Spinup Momentum Error		<= Spinup Momentum Error			
In Eclipse	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	
Sine Sun Elevation				> Coarse Sun Elevation Error	> Coarse Sun Elevation Error	> Coarse Sun Elevation Error	> Coarse Sun Elevation Error
Wheel Spin Rate							
Sine Sun Azimuth							
Tumble Rate, Omega							
New Mode	**Reorient**						

Satellite Control Specification
(Coming From Perspective)

Current Mode	Reorient	Deploy Wheel	Acquire	Acquire	Paddle Deploy	Paddle Deploy	Orbit Day	Orbit Day
Paddles					Deployed	Not Deployed		
Time In Mode	>= Reorient Mode Delay	>= Deploy Wheel Mode Delay	< Acquire Mode Delay	>= Acquire Mode Delay		>= Paddle Deploy Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay
XZ Momentum Error								
Optical System								Tracking
Momentum Error	<= Spinup Momentum Error							
In Eclipse	FALSE		FALSE	FALSE			FALSE	
Sine Sun Elevation	<= Coarse Sun Elevation Error			<= Coarse Sun Elevation Error			<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error
Wheel Spin Rate	>= Nominal Wheel Rate							
Sine Sun Azimuth				> Coarse Sun Azimuth Error			> Coarse Sun Azimuth Error	> Coarse Sun Azimuth Error
Tumble Rate, Omega								
New Mode	**Acquire**							

Satellite Control Specification
(Coming From Perspective)

Current Mode	Reorient	Deploy Wheel
Paddles		
Time In Mode	>= Reorient Mode Delay	< Deploy Wheel Mode Delay
XZ Momentum Error		
Optical System		
Momentum Error	<= Spinup Momentum Error	
In Eclipse	FALSE	
Sine Sun Elevation	<= Coarse Sun Elevation Error	
Wheel Spin Rate	< Nominal Wheel Rate	
Sine Sun Azimuth		
Tumble Rate, Omega		
New Mode	**Deploy Wheel**	

Current Mode	Acquire	Paddle Deploy
Paddles	Not Deployed	Not Deployed
Time In Mode	>= Acquire Mode Delay	< Paddle Deploy Mode Delay
XZ Momentum Error		
Optical System		
Momentum Error		
In Eclipse	FALSE	
Sine Sun Elevation	<= Coarse Sun Elevation Error	
Wheel Spin Rate		
Sine Sun Azimuth	<= Coarse Sun Azimuth Error	
Tumble Rate, Omega		
New Mode	**Paddle Deploy**	

Satellite Control Specification
(Coming From Perspective)

Current Mode	Acquire	Orbit Day	Orbit Day	Orbit Day	Orbit Day	Orbit Night	Orbit Night	Orbit Night
Paddles	Deployed							
Time In Mode	>= Acquire Mode Delay	< Orbit Day Mode Delay	< Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay			
XZ Momentum Error								
Optical System			Tracking	Not Tracking	Tracking			Not Tracking
Momentum Error								
In Eclipse	FALSE	FALSE		FALSE		FALSE	FALSE	FALSE
Sine Sun Elevation	<= Coarse Sun Elevation Error			<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error		> Fine Elevation Error	
Wheel Spin Rate								
Sine Sun Azimuth	<= Coarse Sun Azimuth Error			<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error	> Fine Azimuth Error		
Tumble Rate, Omega								
New Mode	**Orbit Day**							

Satellite Control Specification
(Coming From Perspective)

Current Mode	Spinup	Orbit Day	Orbit Day	Orbit Night
Paddles	Deployed			
Time In Mode		>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	
XZ Momentum Error				
Optical System	Tracking	Tracking	Tracking	Tracking
Momentum Error				
In Eclipse		FALSE		TRUE
Sine Sun Elevation		<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Fine Elevation Error
Wheel Spin Rate				
Sine Sun Azimuth		<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error	<= Fine Azimuth Error
Tumble Rate, Omega				<= Maximum Tumble Rate
New Mode	**Orbit Night**			

Satellite Control Specification
(Going To Perspective)

Current Mode	Wait	
Paddles		
Time In Mode	< Wait Mode Delay	>= Wait Mode Delay
XZ Momentum Error		
Optical System		
Momentum Error		
In Eclipse		
Sine Sun Elevation		
Wheel Spin Rate		
Sine Sun Azimuth		
Tumble Rate, Omega		
New Mode	**Wait**	**Detumble**

Satellite Control Specification
(Going To Perspective)

Current Mode	Detumble		
Paddles			
Time In Mode	< Detumble Mode Delay	>= Detumble Mode Delay	>= Detumble Mode Delay
XZ Momentum Error		> XZ Momentum Error Threshold	<= XZ Momentum Error Threshold
Optical System			
Momentum Error			
In Eclipse			
Sine Sun Elevation			
Wheel Spin Rate			
Sine Sun Azimuth			
Tumble Rate, Omega			
New Mode	**Detumble**	**Detumble**	**Spinup**

Satellite Control Specification
(Going To Perspective)

Current Mode	Spinup						
Paddles	Deployed	Not Deployed		Not Deployed		Not Deployed	
Time In Mode		< Spinup Mode Delay	<Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay
XZ Momentum Error				<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold
Optical System	Tracking		Not Tracking		Not Tracking		Not Tracking
Momentum Error				> Spinup Momentum Error	> Spinup Momentum Error		
In Eclipse						TRUE	TRUE
Sine Sun Elevation							
Wheel Spin Rate							
Sine Sun Azimuth							
Tumble Rate, Omega							
New Mode	**Orbit Night**	**Spinup**	**Spinup**	**Spinup**	**Spinup**	**Spinup**	**Spinup**

Satellite Control Specification
(Going To Perspective)

Current Mode	Spinup (cont.)			
Paddles	Not Deployed		Not Deployed	
Time In Mode	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay	>= Spinup Mode Delay
XZ Momentum Error	> XZ Momentum Error Threshold	> XZ Momentum Error Threshold	<= XZ Momentum Error Threshold	<= XZ Momentum Error Threshold
Optical System		Not Tracking		Not Tracking
Momentum Error			<= Spinup Momentum Error	<= Spinup Momentum Error
In Eclipse			FALSE	FALSE
Sine Sun Elevation				
Wheel Spin Rate				
Sine Sun Azimuth				
Tumble Rate, Omega				
New Mode	**Detumble**	**Detumble**	**Reorient**	**Reorient**

Satellite Control Specification
(Going To Perspective)

Current Mode	Reorient					
Paddles						
Time In Mode		>= Reorient Mode Delay	< Reorient Mode Delay	>= Reorient Mode Delay	>= Reorient Mode Delay	>= Reorient Mode Delay
XZ Momentum Error						
Optical System						
Momentum Error		> Spinup Momentum Error		<= Spinup Momentum Error	<= Spinup Momentum Error	<= Spinup Momentum Error
In Eclipse	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
Sine Sun Elevation				> Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error
Wheel Spin Rate					< Nominal Wheel Rate	>= Nominal Wheel Rate
Sine Sun Azimuth						
Tumble Rate, Omega						
New Mode	**Spinup**	**Spinup**	**Reorient**	**Reorient**	**Deploy Wheel**	**Acquire**

Satellite Control Specification
(Going To Perspective)

Current Mode	Deploy Wheel	
Paddles		
Time In Mode	< Deploy Wheel Mode Delay	>= Deploy Wheel Mode Delay
XZ Momentum Error		
Optical System		
Momentum Error		
In Eclipse		
Sine Sun Elevation		
Wheel Spin Rate		
Sine Sun Azimuth		
Tumble Rate, Omega Omega		
New Mode	**Deploy Wheel**	**Acquire**

Satellite Control Specification
(Going To Perspective)

Current Mode	Acquire					
Paddles					Not Deployed	Deployed
Time In Mode		< Acquire Mode Delay	>= Acquire Mode Delay	>= Acquire Mode Delay	>= Acquire Mode Delay	>= Acquire Mode Delay
XZ Momentum Error						
Optical System						
Momentum Error						
In Eclipse	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
Sine Sun Elevation			<= Coarse Sun Elevation Error	> Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error
Wheel Spin Rate						
Sine Sun Azimuth			> Coarse Sun Azimuth Error		<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error
Tumble Rate, Omega Omega						
New Mode	**Spinup**	**Acquire**	**Acquire**	**Reorient**	**Paddle Deploy**	**Orbit Day**

Satellite Control Specification
(Going To Perspective)

Current Mode	Paddle Deploy		
Paddles	Deployed	Not Deployed	Not Deployed
Time In Mode		>= Paddle Deploy Mode Delay	< Paddle Deploy Mode Delay
XZ Momentum Error			
Optical System			
Momentum Error			
In Eclipse			
Sine Sun Elevation			
Wheel Spin Rate			
Sine Sun Azimuth			
Tumble Rate, Omega Omega			
New Mode	**Acquire**	**Acquire**	**Paddle Deploy**

Satellite Control Specification
(Going To Perspective)

Current Mode	Orbit Day						
Paddles							
Time In Mode		< Orbit Day Mode Delay	< Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay
XZ Momentum Error							
Optical System	Not Tracking		Tracking	Not Tracking	Tracking		Tracking
Momentum Error							
In Eclipse	TRUE	FALSE		FALSE		FALSE	
Sine Sun Elevation				<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	> Coarse Sun Elevation Error	> Coarse Sun Elevation Error
Wheel Spin Rate							
Sine Sun Azimuth				<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error		
Tumble Rate, Omega							
New Mode	**Spinup**	**Orbit Day**	**Orbit Day**	**Orbit Day**	**Orbit Day**	**Reorient**	**Reorient**

Satellite Control Specification
(Going To Perspective)

Current Mode	Orbit Day (cont.)			
Paddles				
Time In Mode	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay	>= Orbit Day Mode Delay
XZ Momentum Error				
Optical System		Tracking	Tracking	Tracking
Momentum Error				
In Eclipse	FALSE		FALSE	
Sine Sun Elevation	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error	<= Coarse Sun Elevation Error
Wheel Spin Rate				
Sine Sun Azimuth	> Coarse Sun Azimuth Error	> Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error	<= Coarse Sun Azimuth Error
Tumble Rate, Omega				
New Mode	**Acquire**	**Acquire**	**Orbit Night**	**Orbit Night**

Satellite Control Specification
(Going To Perspective)

Current Mode	Orbit Night				
Paddles					
Time In Mode					
XZ Momentum Error					
Optical System			Not Tracking	Tracking	Tracking
Momentum Error					
In Eclipse	FALSE	FALSE	FALSE	TRUE	TRUE
Sine Sun Elevation		> Fine Elevation Error		<= Fine Elevation Error	<= Fine Elevation Error
Wheel Spin Rate					
Sine Sun Azimuth	> Fine Azimuth Error			<= Fine Azimuth Error	<= Fine Azimuth Error
Tumble Rate, Omega				> Maximum Tumble Rate	<= Maximum Tumble Rate
New Mode	**Orbit Day**	**Orbit Day**	**Orbit Day**	**Spinup**	**Orbit Night**

Satellite Control Specification
(Going To Perspective)

Current Mode	Ground Control
Paddles	
Time In Mode	> Command Mode Delay
XZ Momentum Error	
Optical System	
Momentum Error	
In Eclipse	
Sine Sun Elevation	
Wheel Spin Rate	
Sine Sun Azimuth	
Tumble Rate, Omega	
New Mode	**Detumble**